Minterest Finance

Security Assessment

March 29, 2022

Prepared for:
Kyn Chaturvedi and Denis Romanovsky
Minterest Finance

Prepared by: **Spencer Michaels, Jaime Iglesias, and Troy Sargent**

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 mutually 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

Minterest Finance engaged Trail of Bits to review the security of its lending platform. From January 10 to January 21, 2022, a team of two consultants conducted a security review of the client-provided source code, with six person-weeks of effort. Details of the project's timeline, test targets, and coverage are provided in subsequent sections of this report.

On March 7, 2022, Trail of Bits consultants conducted a complimentary four-hour review of the fixes implemented by Minterest Finance for issues identified in this report. Minterest Finance addressed nine of the issues, accepted the risks associated with three, identifying those issues as intended behavior, and left three unaddressed; the last two could not be definitively verified as fixed or not fixed. Details of the fix review are provided in appendix H.

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 access to the system's source code and documentation. We performed static and dynamic analysis as well as a manual review of the codebase.

Summary of Findings

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

EXPOSURE ANALYSIS

Severity	*Count*
High	2
Medium	1
Low	6
Informational	4
Undetermined	4

CATEGORY BREAKDOWN

Category	*Count*
Data Validation	9
Denial of Service	3
Undefined Behavior	3
Auditing and Logging	1
Configuration	1

Notable Findings

Significant flaws that impact system confidentiality, integrity, or availability are described below.

- **TOB-MNTR-1**
  If one of the tokens transferred through `MinterestNFT.batchTransfer()` has an `amount` of zero, the bookkeeping process for that token and all subsequent tokens sent through the transaction will be aborted. A malicious user could thus transfer tokens to someone else but remain the beneficiary of their yields.

- **TOB-MNTR-2**
  Liquidation transactions are vulnerable to sandwich attacks, through which an attacker could cause the collateral price to spike by such a significant amount that the liquidation would be aborted. Using this vulnerability, a malicious user could force a liquidator to increase his or her slippage tolerance and to ultimately settle the transaction at a disadvantageous execution price.

- **TOB-MNTR-3**
  An extremely large array is allocated repeatedly within common external functions that are central to the protocol's operation, wasting a significant amount of gas.

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:

Spencer Michaels, Technical Lead
spencer.michaels@trailofbits.com

Troy Sargent, Consultant
troy.sargent@trailofbits.com

Jaime Iglesias, Consultant
jaime.iglesias@trailofbits.com

Project Timeline

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

Date	Event
January 3, 2022	Pre-project kickoff call
January 11, 2022	Status update meeting #1
January 18, 2022	Status update meeting #2
January 25, 2022	Report readout meeting; delivery of report draft
February 2, 2022	Delivery of final report
February 11, 2022	Receipt of fixes from the client
March 7, 2022	Fix review
March 8, 2022	Delivery of fix review
March 29, 2022	Delivery of public report

Project Goals

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

- Is Minterest's novel liquidation scheme—which is intended to give the protocol a competitive advantage—technically and economically sound?
- Could a malicious borrower avoid a timely liquidation of under-collateralized assets or gain another unfair advantage?
- Are the incentives and disincentives for borrowers and liquidators enforced in accordance with Minterest's business model?
- Is the ownership of NFTs tracked and enforced correctly?
- Are gas costs, especially those for liquidations, kept to a reasonable level?
- Are the Minterest contracts amenable to auditing (e.g., do they execute thorough event logging and have thorough, up-to-date documentation and comments)?

Project Targets

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

Minterest Finance Protocol

Repository	https://github.com/minterest-finance/protocol
Version	`c56d0e72b39f3c6da5256c15718a1c9a97bde3a5`
Type	Blockchain Application
Platform	Solidity

Project Coverage

This section provides an overview of the analysis coverage of the review, as determined by our high-level engagement goals. Our approaches and their results include the following:

- A review of the liquidation functionality found that large array allocations (TOB-MNTR-3) and the use of long or nested loops (appendix D) carry excessive gas costs.
- A review of the lending / borrowing functionality found a way in which a user could borrow funds from his or her own collateral (TOB-MNTR-14).
- Dynamic analysis confirmed that users incur high gas costs during liquidations (appendix C).
- A review of the external functions for front-running opportunities found that an attacker could cause a liquidation to be aborted by launching a sandwich attack (TOB-MNTR-2).
- A review of NFTs and emission boosts revealed that the NFT bookkeeping could become out of sync with the actual state of token ownership (TOB-MNTR-3).
- A review of the use of third-party price oracles found minor input validation issues.
- A review of the proxy contract revealed concerns surrounding the use of the current implementation in conjunction with an implementation that inherits from the `AccessControl` contract (TOB-MNTR-4).
- A review of the project's access controls revealed that Minterest Finance holds substantial administrative powers and can freely transfer MNT tokens from the `Supervisor` contract.

Automated Testing Results

Trail of Bits has developed unique tools for testing smart contracts. Details on the tool used in this project are provided below.

- Slither is a static analysis framework that can statically verify algebraic relationships between Solidity variables. We used Slither to verify that a borrower cannot take advantage of a reentrancy to borrow additional funds while still repaying a debt, thereby stealing funds (see appendix F).

Automated testing techniques augment our manual security review but do not replace it. Each technique has limitations: Slither, for instance, may identify security properties that fail to hold when Solidity is compiled to EVM bytecode.

Our automated testing and verification focused on the following system property:

Reentrant calls cannot be used to cause erroneous accounting. When a user takes out or repays a loan, the user's debt balance, the `accountBorrows` value, is updated. If an outdated `accountBorrows` value were read during an external call, or if the value were overwritten, the protocol could be vulnerable to theft. Thus, we verified that users cannot make reentrant calls to functions that update their debt balances. To do so, we checked that those functions are protected by a mutex, `nonReentrant`.

Property	Tool	Script
Users cannot reenter the functions that write to `accountBorrows`.	Slither	Appendix F

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	Parts of the codebase lack overflow protections.	Moderate
Auditing	Certain critical functions do not log events.	Moderate
Authentication / Access Controls	Although Minterest's proxy implementation is fully functional, we identified some architectural concerns.	Moderate
Complexity Management	The complexity of the protocol, especially with regard to liquidations, results in many gas-heavy operations and long loops throughout the codebase, reducing the viability of Minterest's unique liquidation scheme.	Weak
Cryptography and Key Management	The protocol does not use keys or perform encryption.	Not Applicable
Decentralization	Minterest Finance holds all administrative power over the protocol, and the `Supervisor` contract's owner can freely withdraw MNT from the system.	Moderate
Documentation	The general workings of the protocol are well documented, but many comments misrepresent its business logic.	Moderate
Front-Running Resistance	The design of the liquidation process renders it inherently vulnerable to sandwich attacks; the process will likely require significant reworking.	Weak
Low-Level Manipulation	The protocol does not make any low-level calls.	Not Applicable

Testing and Verification	The Minterest codebase has 100% unit test coverage.	**Satisfactory**

Summary of Findings

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

ID	Title	Type	Severity
1	MinterestNFT batched transfers can cause bookkeeping errors	Data Validation	High
2	Risk of a systemic liquidation failure due to a denial of service	Denial of Service	High
3	Excessive gas costs due to the repeated allocation of a large array	Denial of Service	Medium
4	Risks associated with using MintProxy alongside contracts that implement AccessControl	Configuration	Low
5	Error-prone empty function	Undefined Behavior	Low
6	Inconsistent definition of total supply	Undefined Behavior	Low
7	Input validation of Chainlink oracle	Data Validation	Low
8	setAdmin does not emit events	Auditing and Logging	Low
9	Block-time assumption may cause interest-calculation discrepancies	Data Validation	Low
10	Market-addition costs increase linearly	Denial of Service	Informational
11	Incorrect check of whether a contract is an ERC20	Data Validation	Informational
12	Block number overflow	Undefined Behavior	Informational
13	Potential reentrancy vulnerability in repayBorrow	Data Validation	Informational

14	Users can borrow assets they are actively using as collateral	Data Validation	**Undetermined**
15	Unbounded loop could enable a denial of service	Data Validation	**Undetermined**
16	Stable parameter is not guaranteed to be a stablecoin	Data Validation	**Undetermined**
17	autoLiquidationRepayDeadBorrow's lack of data validation	Data Validation	**Undetermined**

Detailed Findings

1. MinterestNFT batched transfers can cause bookkeeping errors	
Severity: **High**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-MNTR-1
Target: `contracts/MinterestNFT.sol:504`	

Description

If the set of tokens passed to the `batchTransfer` function includes one token with an `amount` of zero, the transfer may result in bookkeeping errors.

`MinterestNFT` follows the ERC1155 standard and uses the OpenZeppelin reference implementation. This implementation defines an empty hook that is executed before certain functions are called. Implementers can override this hook to add custom logic for transfer, minting, and burning operations (including batched versions of these operations).

```
function _beforeTokenTransfer(
    address operator,
    address from,
    address to,
    uint256[] memory ids,
    uint256[] memory amounts,
    bytes memory data
) internal virtual {}
```

Figure 1.1: `openzeppelin-contracts/contracts/ERC1155/ERC1155.sol:405-412`

The Minterest implementation overrides this hook with custom logic for keeping a list of all tokens owned by a given account.

```
function _beforeTokenTransfer(
    address,
    address from,
    address to,
    uint256[] memory ids,
    uint256[] memory amounts,
    bytes memory
) internal virtual override {
    if (from == address(0x0)) {
```

```
            _accountToTokenIds[to].push(ids[0]);
            return;
        }

        if (to == address(0x0)) {
            // TODO
        }

        updateMntStateIfNeeded(from, to, ids);

        for (uint256 j = 0; j < ids.length; j++) {
            if (amounts[j] == 0) {
                return;
            }

            uint256[] memory ownedTokenIds = _accountToTokenIds[from];
            delete _accountToTokenIds[from];

            for (uint256 i = 0; i < ownedTokenIds.length; i++) {
                if (ownedTokenIds[i] == ids[j]) {
                    if (balanceOf(to, ownedTokenIds[i]) == 0) {
                        _accountToTokenIds[to].push(ownedTokenIds[i]);
                    }

                    if (balanceOf(from, ownedTokenIds[i]) - amounts[j] > 0) {
                        _accountToTokenIds[from].push(ownedTokenIds[i]);
                    }

                    continue;
                }

                _accountToTokenIds[from].push(ownedTokenIds[i]);
            }
        }
    }
```

Figure 1.2: `protocol/contracts/MinterestNFT.sol:483-526`

The `_beforeTokenTransfer()` function loops through all of the tokens being transferred, performing accounting beyond that in the reference implementation.

In the case of a batched transfer in which the `amount` of a token is zero, the loop is terminated by a `return` statement.

```
    …
    for (uint256 j = 0; j < ids.length; j++) {
          if (amounts[j] == 0) {
              return;
          }
       …
     }
```

Figure 1.3: `protocol/contracts/MinterestNFT.sol:502-505`

As a result, the bookkeeping logic in the hook will not be executed for subsequent tokens. This will ultimately lead to a discrepancy in the token contract's internal bookkeeping: `_accountToTokenIds` will indicate that the tokens are still owned by their original owner, while `_balances` will identify the owner as the address to which the tokens were transferred.

Exploit Scenario

Alice owns a handful of NFTs that allow her to obtain a boost on her yields. She decides to transfer the NFTs to a different account that she controls and calls `batchTransfer` with zero as the first `amount`. When the transfer is complete, she puts the NFTs up for sale.

Eve buys the NFTs. She then realizes that even though she owns them, she will not receive any yield on them, as Alice is still the beneficiary of the yield.

Recommendations

Short term, change the aforementioned `return` statement to a `continue` statement so that the hook code continues looping even if one of the tokens being transferred has an `amount` of zero.

Long term, review all assumptions about the effects of the hook on token bookkeeping.

2. Risk of a systemic liquidation failure due to a denial of service

Severity: **High**	Difficulty: **Medium**
Type: Denial of Service	Finding ID: TOB-MNTR-2
Target: `contracts/Liquidation.sol`	

Description

To complete a liquidation, the protocol must trade the collateral asset for the debt asset and then distribute the proceeds to liquidity providers. This dependence on a successful trade creates a denial-of-service attack vector, as an attacker could observe the mempool and cause a systemic liquidation failure.

Specifically, because Uniswap V2-like automated market makers (AMMs) are vulnerable to temporary price manipulations, an attacker could cause the price of a trade to exceed the liquidator's slippage tolerance by front-running the liquidator's swap. As a result, the price of the assets in the pool would change, and the trade would fail. This process could be used in a sandwich attack, in which a bot would then buy a large amount of the desired asset and increase the price paid by the protocol to settle a trade.

When a liquidation requires a trade (i.e., either `seizeAmount` or `repayAmount` is non-zero), `swapExactTokensForTokens` is called. The `amountOutMin` parameter specifies the minimum number of tokens that the protocol is willing to purchase at the *current* oracle price.

```
 if (seizeAmount > 0) {
    address market = marketAddresses[i];
    IERC20 marketUnderlying = MToken(market).underlying();
    uint256 amountIn =
seizeAmount.mul(expScale).div(oracle.getUnderlyingPrice(MToken(market)));
    uint256 amountOutMin =
seizeAmount.mul(expScale.sub(swapFeeMantissa.add(slippages[i]))).div(
        oracle.getUnderlyingPrice(MToken(address(stable)))
    );
    address[] memory path = new address[](2);
    path[0] = address(marketUnderlying);
    path[1] = address(stable);
    marketUnderlying.safeIncreaseAllowance(address(swapRouter), amountIn);
    swapRouter.swapExactTokensForTokens(amountIn, amountOutMin, path, address(this),
block.timestamp + 30);
```

Figure 2.1: Part of the `doSwap` function

If the price of an asset in the AMM pool deviates too far from the quoted oracle price, the protocol will be unable to purchase the minimum number of tokens (specified by `amountOutMin`). The transaction will then revert, and the liquidation will not be completed (figure 2.2). If additional liquidations fail, the protocol will be unable to prevent further loan insolvency, which will make providing liquidity less profitable.

Note that although a liquidity provider can increase his or her slippage tolerance, doing so increases the profitability of such an attack and decreases the amount of proceeds paid out to liquidity providers. In fact, attackers engage in this kind of griefing to extract the spread between the current price and the maximum price that the protocol is willing to pay.

```
require(amounts[amounts.length - 1] >= amountOutMin, 'UniswapV2Router:
INSUFFICIENT_OUTPUT_AMOUNT');
```

Figure 2.2: Part of the `swapExactTokensForTokens` function

Exploit Scenario

An attacker watches the Moonbeam network's transaction mempool for liquidation attempts on the Minterest protocol. When the attacker sees a transaction sent by a liquidator, the attacker performs a swap through the same AMM pool. The attack proceeds as follows:

1. The attacker front-runs the liquidation transaction, causing the price to exceed the slippage tolerance.
2. Because `amountOutMin` is not satisfied, the transaction reverts and fails.
3. The attacker engages in back-running, trading in the opposite direction and restoring the price.

The liquidator is forced to increase his or her slippage tolerance to complete liquidations. Following the increase, the attacker adopts a different strategy:

1. The attacker engages in front-running to move the price to the maximum slippage tolerance.
2. The trade is settled at an execution price disadvantageous to the liquidator, but the liquidation is successful.
3. The attacker, through back-running, trades in the opposite direction and profits off of the spread.

Recommendations

Short term, allow liquidators to transfer debt assets directly to the protocol rather than relying on an external market.

Long term, assess the viability of the liquidation mechanism. Designing a system to limit the maximal extractable value (MEV) often changes the profitable MEV opportunities available to an attacker, but it does not eliminate them. As described above, an attacker could exploit an MEV opportunity not by running a liquidation bot but by running a sandwich bot.

References

- DeFi Sandwich Attacks
- Frontrunning, Transaction Reordering, and Consensus Instability in Decentralized Exchanges

3. Excessive gas costs due to the repeated allocation of a large array

Severity: **Medium**	Difficulty: **Low**
Type: Denial of Service	Finding ID: TOB-MNTR-3
Target: `contracts/MinterestNFT.sol#L603`	

Description

The `MinterestNFT` contract uses the `createBoostSegments()` function to compute the average boost value. In performing this calculation, the function allocates an extremely large array of booleans, with a size equal to the current block number. This array is populated only sparsely: the final number of entries set to `true` is at most about twice the number of NFT-owning accounts tracked by the `Supervisor` contract. This means that most of the allocated space is unused.

There are currently almost 1.3 million blocks on the Moonbeam network. Assuming maximally efficient packing (i.e., booleans packed consecutively, with one bit for each value), an array of that size would take up about 15,000 256-bit words of space. Since the gas cost of allocating an array above a relatively small size increases exponentially with each word allocated, the operation would consume a very large amount of gas.

Furthermore, the `createBoostSegments()` function is called by `MinterestNFT.calculateEmissionBoost()`, which is called in `Supervisor.distributeSupplierMnt()`. This latter function is invoked by several common external methods, such as `Supervisor.transferAllowed()`, sometimes multiple times per external call. As a result, the abovementioned array allocation is likely to occur very frequently, wasting a considerable amount of gas.

```
603     bool[] memory blocksIndicators = new bool[](currentBlock + 1);
...
609     blocksIndicators[userLastUpdatedBlock] = true;
610     blocksIndicators[currentBlock] = true;
...
612     // For each account NFT token
613     for (uint256 i = 0; i < accountTokens.length; i++) {
614         uint256 tier = _idToTier[accountTokens[i]];
615         uint32 start = tiers[tier].startEmissionBoostBlock;
616         uint32 end = tiers[tier].endEmissionBoostBlock;
617
618         if (isTierActive(tier)) {
619             if (userLastUpdatedBlock <= start && start <= currentBlock &&
```

```
!blocksIndicators[start]) {
                    userPoints[tier].index = getMarketSpecificData(market, tier,
isSupply).startIndex;
                    userPoints[tier].block = start;
                    blocksIndicators[start] = true;
                }
                if (userLastUpdatedBlock <= end && end <= currentBlock &&
!blocksIndicators[end]) {
                    userPoints[userPoints.length - 1 - tier].index =
getMarketSpecificData(market, tier, isSupply)
                        .endIndex;
                    userPoints[userPoints.length - 1 - tier].block = end;
                    blocksIndicators[end] = true;
                }
            }
        }
```

Figure 3.1: The `createBoostSegments()` function in `contracts/MinterestNFT.sol#L603-28`

Recommendations

Short term, reduce the size of the allocated array to only what is necessary, and calculate the average boost value less frequently.

Long term, carefully audit the codebase for large allocations, long or nested loops, and other patterns that consume excessive amounts of gas.

References

- Moonriver Moonbeam Explorer
- Ethereum Yellow Paper

4. Risks associated with using MintProxy alongside contracts that implement AccessControl	
Severity: **Low**	Difficulty: **Low**
Type: Configuration	Finding ID: TOB-MNTR-4
Target: `contracts/MintProxy.sol`	

Description

If the current proxy, `MintProxy`, is used in conjunction with an implementation contract that inherits from the `AccessControl` contract, changing the proxy contract's admin will also change the implementation contract's admin.

The current implementation of `AccessControl` uses the unstructured storage pattern to store the address of its admin. Its use of the pattern is very unusual, since this type of pattern is typically used in proxy contracts to prevent storage clashes between a proxy and the implementation.

```
    /**
     * @dev Storage slot with the admin of the contract.
     * This is the keccak-256 hash of "eip1967.proxy.admin" subtracted by 1.
     */
    bytes32 internal constant _ADMIN_SLOT =
0xb53127684a568b3173ae13b9f8a6016e243e63b6e8ee1178d6a717850b5d6103;
```

Figure 4.1: Part of the `AccessControl` contract

The `MintProxy` contract uses the same storage slot to store the address of its admin.

```
    /**
     * @dev Storage slot with the admin of the contract.
     * This is the keccak-256 hash of "eip1967.proxy.admin" subtracted by 1, and is
     * validated in the constructor.
     */
    bytes32 internal constant _ADMIN_SLOT =
0xb53127684a568b3173ae13b9f8a6016e243e63b6e8ee1178d6a717850b5d6103;
```

Figure 4.2: Part of the `MintProxy` contract

Furthermore, `MintProxy` partially follows the transparent proxy pattern, which means that *some* transactions made by the proxy's admin will still be delegated to the implementation contract.

```
    /**
     * @dev Modifier used internally that will delegate the call to the implementation
unless the sender is the admin.
     */
    modifier ifAdmin() {
        if (msg.sender == _getAdmin()) {
            _;
        } else {
            _fallback();
        }
    }
```

Figure 4.3: Part of the `MintProxy` contract

The intention behind this architecture seems to be enabling the use of the same admin for both the proxy and implementation contracts while maintaining the ability to delegate admin-only operations to the implementation contract. However, the architecture is confusing and can be error-prone.

For example, if the proxy implementation became a fully transparent proxy, admin transactions would not be delegated to the implementation contract; as a result, all of the admin-only functions in the implementation would be unreachable.

Exploit Scenario

The `MintProxy` contract is changed to follow the implementation of a transparent proxy. The proxy is deployed in conjunction with a contract that inherits from `AccessControl`, rendering all admin-only functions in the implementation contract unreachable.

Recommendations

Short term, use different addresses for the admins of the proxy and implementation contracts. Additionally, review the `AccessControl` implementation and consider simply storing addresses rather than using unstructured storage. That change would also require the use of an initializer function in the implementation contract, which could create risks such as initialization front-running.

Alternatively, use a different storage slot for the address of the `MintProxy` contract's admin to avoid using the same slot that `AccessControl` uses for that address.

Lastly, thoroughly document all risks associated with the current implementation of the proxy. For example, following a partially transparent proxy pattern and allowing the proxy's owner to delegate transactions to the implementation contract may result in errors if the current implementation is changed. Furthermore, functions in non-transparent proxies are susceptible to function selector clashes, which can occur when a function in an implementation contract has the same selector as one in the proxy contract.

Long term, review the assumptions about proxy contracts and reconsider whether it is necessary to use the error-prone proxy pattern.

References

- Contract upgrade anti-patterns
- Breaking Aave Upgradeability
- Slither Upgradeability Checks

5. Error-prone empty function	
Severity: **Low**	Difficulty: **Low**
Type: Undefined Behavior	Finding ID: TOB-MNTR-5
Target: `contracts/Governance/Mnt.sol`	

Description
The MNT ERC20 token overrides the `_burn` function but does not implement a replacement. This pattern could result in undefined behavior if a developer assumed the function would burn tokens and attempted to call it. It is best practice to provide a function implementation.

Exploit Scenario
A developer calls `mnt.burn` and prematurely changes his or her contract's record of the amount of MNT that a user has burned. Rather than reverting, the function immediately returns without executing any code. As a result, the contract's state is updated even though no tokens were burned, throwing off the smart contract's internal bookkeeping.

Recommendations
Short term, implement the `burn` function. Use `revert()` in the function's body if it should revert rather than returning.

Long term, review and clearly document the functions to prevent undefined behavior.

6. Inconsistent definition of total supply	
Severity: **Low**	Difficulty: **Low**
Type: Undefined Behavior	Finding ID: TOB-MNTR-6
Target: `contracts/Governance/Mnt.sol`	

Description

In a comment, the MNT ERC20 token specifies a total supply of 100 million. By contrast, the code defines `TOTAL_SUPPLY` as `100_000_030e18`, which is equivalent to 100,000,030 (the exponent represents the scale factor of the fixed-point number). The code and the comment should align with each other.

```
contract Mnt is ERC20, ERC20Permit, ERC20Votes {
    /// @notice Total number of tokens in circulation
    uint256 internal constant TOTAL_SUPPLY = 100_000_030e18; // 100 million MNT

    constructor(address account) ERC20("Minterest", "MNT") ERC20Permit("Minterest") {
        _mint(account, uint256(TOTAL_SUPPLY));
    }
```

Figure 6.1: Part of the `Mnt` contract

Recommendations

Short term, standardize the total supply value in the code and comment.

Long term, ensure that code comments accurately reflect the most recent implementation.

7. Input validation of Chainlink oracle

Severity: **Low**	Difficulty: **Undetermined**
Type: Data Validation	Finding ID: TOB-MNTR-7
Target: `contracts/Oracles/ChainlinkPriceOracle.sol`	

Description

The protocol requires that the quoted oracle price in `convertReportedPrice` be greater than zero and implements a `timestampThreshold` value to enforce the use of recent data. However, the check of whether the `updatedAt` value is less than the current block timestamp is unnecessary; this is because the subsequent operation, `block.timestamp - updatedAt`, will revert if it is not (since solc 0.8.0 implements overflow checks by default).

Instead, the protocol could require that the round in which the price was calculated (the `answeredInRound` value) be equal to the current round ID (`roundID`). This would guarantee that if multiple rounds occurred within the `timestampThreshold` period, data from the most recent round would be used.

```
    (, int256 answer, , uint256 updatedAt, ) = config.chainlinkAggregator.latestRoundData();

    // solhint-disable-next-line not-rely-on-time
    require(updatedAt <= block.timestamp, "Incorrect timestamp");
    require(block.timestamp - updatedAt <= timestampThreshold, "Oracle price expired");

    uint256 convertedPrice = convertReportedPrice(config, answer);
    return convertedPrice.mul(1e28).div(config.underlyingTokenDecimals);
}
```

Figure 7.1: Part of the `getUnderlyingPrice` function

Recommendations

Short term, remove the unnecessary operation, `require(updatedAt <= block.timestamp, "Incorrect timestamp")`, and add `require(answeredInRound == roundId)`.

Long term, keep up to date on Chainlink best practices and the default behavior of the Solidity compiler being used.

8. setAdmin does not emit events

Severity: **Low**	Difficulty: **Undetermined**
Type: Auditing and Logging	Finding ID: TOB-MNTR-8
Target: `contracts/AccessControl.sol`	

Description

Several contracts inherit from `AccessControl` and call `setAdmin` to give an address control over privileged actions; however, `setAdmin` does not follow the best practice of emitting events when important variables are set or changed.

```
function setAdmin(address newAdmin) internal {
    require(newAdmin != address(0), ERR_INVALID_INPUT);
    StorageSlot.getAddressSlot(_ADMIN_SLOT).value = newAdmin;
}
```

Figure 8.1: The `setAdmin` function

Recommendations

Short term, have `setAdmin` emit an event when it sets an admin.

Long term, review critical operations in the codebase and ensure that relevant information is consistently logged.

9. Block-time assumption may cause interest-calculation discrepancies	
Severity: **Low**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-MNTR-9
Target: `contracts/KinkMultiplierModel.sol`	

Description

To calculate the amount of interest paid out to liquidity providers, Minterest uses an estimate of the number of blocks mined annually (`blocksPerYear`). This calculation assumes a block time of roughly 15 seconds, or 2,102,400 blocks per year. However, because this assumption is based on the Ethereum network, which has a different block time from the Moonbeam network, it is inaccurate.

The Moonbeam documentation states that the current block time is six seconds, which means that more than twice as many blocks are mined on Moonbeam as on Ethereum. Thus, liquidity providers on the Moonbeam network will earn more than twice the amount of interest that they would earn in the same time period on Ethereum.

```
uint256 public constant blocksPerYear = 2102400;
```

Figure 9.1: Part of the `KinkMultiplierModel` contract

Recommendations

Short term, assess the block-time assumption and update it to reflect the network on which Minterest will be deployed.

Long term, ensure that all network-specific values are correct, and consider using `block.timestamp` instead of a hard-coded value to calculate accrued interest.

References

- Moonbeam FAQ

10. Market-addition costs increase linearly	
Severity: **Informational**	Difficulty: **Low**
Type: Denial of Service	Finding ID: TOB-MNTR-10
Target: `contracts/Supervisor.sol`	

Description

When a new market is added to the protocol, the `_supportMarket` function searches all existing markets to prevent the addition of a duplicate market; however, a duplicate market would cause the function call to revert, because the call would violate an invariant requiring that the new market be one that is not already listed.

Each market added to the protocol increases the length of the array that the function must search, which has no upper bound. Since the property `isListed` can never be set back to `false`, it is not possible to add a duplicate market. Thus, this `for` loop results in unnecessary gas costs and can be removed.

```
function _supportMarket(MToken mToken) external adminOnly {
    require(mToken.supportsInterface(type(MTokenInterface).interfaceId),
ERR_CONTRACT_DOES_NOT_SUPPORT_INTERFACE);
    require(!markets[address(mToken)].isListed, ERR_MARKET_ALREADY_LISTED);

    for (uint256 i = 0; i < allMarkets.length; i++) {
        require(allMarkets[i] != mToken, ERR_MARKET_ALREADY_ADDED);
    }

    markets[address(mToken)].isListed = true;
    markets[address(mToken)].utilizationFactorMantissa = 0;
    markets[address(mToken)].liquidationFeeMantissa = 0;
    allMarkets.push(mToken);

    emit MarketListed(mToken);
}
```

Figure 10.1: The `_supportMarket` function

Recommendations

Short term, remove the unnecessary `for` loop.

Long term, before using computationally intensive code to prevent a certain condition (e.g., the addition of a duplicate market), confirm whether that condition is possible.

11. Incorrect check of whether a contract is an ERC20

Severity: **Informational**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-MNTR-11
Target: `contracts/MToken.sol`	

Description

When an MToken contract is constructed, it tries to verify that the underlying token is an ERC20 contract by calling `totalSupply`. As of solc 0.5.0, when a function declares a return type, a check is performed to ensure that the return value is non-empty; if the return value is empty, the transaction will revert. However, this does not guarantee that the contract implements the ERC20 standard. A fake ERC20 that only defines `totalSupply` could therefore be added as an MToken.

```
underlying_.totalSupply();
```

Figure 11.1: Part of the MToken contract

Exploit Scenario

A fake ERC20 contract is added as an MToken because it implements `totalSupply`. However, the contract does not implement other ERC20 methods and causes undefined behavior.

Recommendations

Short term, remove this return value check, which does not provide a security guarantee and allows the creation of markets with fake ERC20 contracts.

Long term, review the Token Integration Checklist and develop a due diligence process for auditing tokens before adding them to the Minterest protocol.

12. Block number overflow

Severity: **Informational**	Difficulty: **High**
Type: Undefined Behavior	Finding ID: TOB-MNTR-12
Target: `contracts/MinterestNFT.sol:370` `contracts/Supervisor.sol:755-771`	

Description

The `MinterestNFT` contract's `enableTiers()` function uses `uint32`-typed variables to store the current block number—the original type of which is `uint256`. This could cause unexpected behavior if the block number approached a multiple of the maximum `uint32` value. Although that is extremely unlikely to occur on the Moonbeam network in the near future, it does pose a problem for the long-term resiliency of the Minterest protocol.

```
368    function enableTiers(uint256[] memory tiersForEnabling) public
adminOr(address(whitelist)) {
369        require(whitelist != Whitelist(address(0)), "Address <...> can not be zero");
370        uint32 currentBlock = uint32(getBlockNumber());
```

Figure 12.1: `contracts/MinterestNFT.sol:368-370`

Given the subsequent operations performed in `enableTiers`, a block number overflow could cause a wraparound, rendering the end position of a tier's range greater than the start of the range. This would lead to unexpected reverts.

```
378    require(
379        currentBlock < tiers[tier].endEmissionBoostBlock,
380        "MinterestNFT: The end block must be larger than the current"
381    );
```

Figure 12.2: `contracts/MinterestNFT.sol:378-381`

Alternatively, if the current block number modulo `UINT32_MAX` approached the range's original start value, expired ranges could become valid again.

Recommendations

Short term, store all block numbers in `uint256`-typed variables to prevent overflows.

Long term, carefully audit any conversions from larger to smaller integer types.

13. Potential reentrancy vulnerability in repayBorrow

Severity: **Informational**	Difficulty: **Undetermined**
Type: Data Validation	Finding ID: TOB-MNTR-13
Target: `contracts/Supervisor.sol`	

Description

The Minterest protocol prevents a reentrancy vulnerability by using a global mutex on functions with the `nonReentrant` modifier, such as `repayBorrow`. Without this mutex, an attacker could invoke `doTransferIn` on an ERC777 token (or an ERC20 token with hooks) and use the token's callback function to borrow additional tokens while still repaying his or her debt. Because the attacker's debt balance would not be updated correctly, the attacker could use this process to steal funds.

We set the severity of this issue to informational because the current implementation includes a mitigation for the attack vector. We have also provided a Slither script (appendix F) that can be run in the CI pipeline to ensure that the code does not regress and render the reentrancy vulnerability exploitable.

```
uint256 borrowBalance = borrowBalanceStoredInternal(borrower);

if (repayAmount == type(uint256).max) {
    repayAmount = borrowBalance;
}
[...]
actualRepayAmount = doTransferIn(payer, repayAmount);
[...]
uint256 accountBorrowsNew = borrowBalance.sub(actualRepayAmount);
uint256 totalBorrowsNew = totalBorrows.sub(actualRepayAmount);

accountBorrows[borrower].principal = accountBorrowsNew;
```

Figure 13.1: Part of the `repayBorrow` function

Exploit Scenario

A developer optimizes the Minterest protocol, which has listed an ERC777 token as an MToken, and removes the `nonReentrant` modifier. After taking out a loan, an attacker calls `repayBorrow` on the ERC777 token, the callback function of which calls `borrow`. This call increases the attacker's debt. However, because `borrowBalance` is stored in memory, it does not reflect the change. Only after the transfer is complete is the attacker's principal updated in storage, enabling the attacker to steal the amount borrowed through the reentrant call.

Recommendations

Short term, integrate the Slither check in appendix F into Minterest's CI pipeline to prevent a regression from leaving the codebase vulnerable to this reentrancy.

Long term, refactor the accounting code to follow the checks-effects-interactions pattern and avoid onboarding ERC777 tokens without first assessing the related risks.

References

- "7.6 Reentrancy Checks Are Necessary" (section of the ChainSecurity cToken Audit)

14. Users can borrow assets they are actively using as collateral

Severity: **Undetermined**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-MNTR-14
Target: `contracts/Supervisor.sol:275-289`	

Description

When a user calls `MToken.borrow()` to borrow tokens from a market, the internal function `Supervisor.borrowAllowed()` verifies that the user is allowed to perform the borrow operation. However, this latter function does not check whether the user is borrowing the same type of asset as the collateral he or she has supplied. In other words, a user can borrow tokens from the collateral that the same user has supplied.

The Minterest protocol prohibits users from borrowing assets worth more than the collateral they have provided, so a user cannot directly exploit this issue to borrow more funds than he or she should be able to borrow. However, a user *can* borrow the vast majority of his or her collateral to continue accumulating MNT rewards while largely avoiding the risks of providing collateral.

```
275    function borrowAllowed(address mToken, address borrower, uint256 borrowAmount)
...      external override {
280        // Bells and whistles to notify user - operation is paused.
281        require(!borrowKeeperPaused[mToken], ERR_OPERATION_PAUSED);
282        require(markets[mToken].isListed, ERR_MARKET_NOT_LISTED);
283
284        if (!markets[mToken].accountMembership[borrower]) {
285            // only mTokens may call borrowAllowed if borrower not in market
286            require(msg.sender == mToken, ERR_INVALID_SENDER);
287
288            // attempt to enable market for the borrower
289            enableMarketAsCollateralInternal(MToken(msg.sender), borrower);
```

Figure 14.1: `contracts/Supervisor.sol#L275-L289`

Exploit Scenario

An attacker provides 10 ETH to the protocol as collateral and then immediately borrows 9 ETH. He continues to earn MNT rewards on his collateral but retains the use of most of the collateral. The attacker, through flash loans, could also resupply the borrowed amount as collateral and then immediately take out another loan, repeating the process until the amount borrowed asymptotically approached the amount of liquidity provided.

Recommendations

Short term, determine whether borrowers' ability to borrow their own collateral is an issue. (Note that the front end of Minterest Finance competitor Compound disallows such operations, but its actual contracts do not.) If it is, have `Supervisor.borrowAllowed()` check whether a user is attempting to borrow the same asset that he or she has staked as collateral and block the operation if so.

Long term, assess whether the liquidity-mining incentives accomplish their intended purpose.

15. Unbounded loop could enable a denial of service

Severity: **Undetermined**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-MNTR-15
Target: `contracts/Liquidation.sol:210-211`	

Description

The `liquidateUnsafeLoan()` function loops over each market that a borrower has entered. Since there is no apparent limit on the number of markets a borrower can participate in simultaneously, the amount of gas required to liquidate the position of a borrower in numerous markets could exceed that allotted for the transaction, preventing a complete liquidation. However, there may not be enough markets for this issue to occur.

```
200    function liquidateUnsafeLoan(
201       address borrower,
202       uint256 drRateMantissa,
203       uint256[] memory slippages,
204       IERC20 stable
205    ) external onlyTrustedLiquidators nonReentrant {
206       AccountLiquidationAmounts memory vars;
207
208       require(drRateMantissa <= expScale, "Incorrect drRateMantissa");
209
210       supervisor.accrueAutoLiquidation(borrower);
211       vars = calculateLiquidationAmounts(borrower, drRateMantissa, safetyBuffer);
```

Figure 15.1: `contracts/Liquidation.sol:200-211`

Recommendations

Short term, determine whether this issue poses a serious risk to the protocol. In making this decision, consider the typical amount of gas consumed per market during a liquidation and the approximate number of markets that Minterest intends to enable (i.e., the number of markets that a borrower will be able to participate in simultaneously).

Long term, carefully audit operations that consume a large amount of gas, especially those in loops.

16. Stable parameter is not guaranteed to be a stablecoin

Severity: **Undetermined**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-MNTR-16
Target: `contracts/Liquidation.sol`	

Description

The liquidation functions have a parameter, `stable`, that is presumably the address of an ERC20 stablecoin; when a liquidation requires it, collateral is traded for the token through a call to `doSwap`. However, `stable` can be set to any token for which Minterest has a configured Chainlink oracle. Thus, a liquidator can pass in any non-stablecoin ERC20 token, causing the treasury to hold that token instead of a stablecoin. If Minterest ever enables external parties to serve as liquidators, it should implement validation of the token being swapped rather than allowing the liquidator to control the argument.

```
    function liquidateUnsafeLoan(
        address borrower,
        uint256 drRateMantissa,
        uint256[] memory slippages,
        IERC20 stable
     ) external onlyTrustedLiquidators nonReentrant {
```

Figure 16.1: Part of the `liquidateUnsafeLoan` function

```
uint256 amountOutMin = seizeAmount.mul(expScale.sub(swapFeeMantissa.add(slippages[i]))).div(
    oracle.getUnderlyingPrice(MToken(address(stable)))
);
address[] memory path = new address[](2);
path[0] = address(marketUnderlying);
path[1] = address(stable);
marketUnderlying.safeIncreaseAllowance(address(swapRouter), amountIn);
swapRouter.swapExactTokensForTokens(amountIn, amountOutMin, path, address(this),
block.timestamp + 30);
```

Figure 16.2: Part of the doSwap function

Recommendations

Short term, determine which ERC20 tokens liquidators should be allowed to swap and add validation accordingly.

Long term, be mindful of the fact that allowing users to control variables such as the output token in a swap may facilitate malicious behavior.

17. autoLiquidationRepayDeadBorrow's lack of data validation	
Severity: **Undetermined**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-MNTR-17
Target: `contracts/MToken.sol`	

Description

The `autoLiquidationRepayDeadBorrow` function enables a liquidator to use the protocol's interest to pay off a loan. However, it does not verify that a loan is eligible for liquidation or implement any safeguards. The Minterest team anticipates that this function will be called only by trusted actors, but it is possible for a liquidator to take out a loan and effectively cancel his or her own debt. Essentially, liquidity providers are not given strong guarantees regarding when this function will be used and how it will affect their earnings.

Additionally, because the function is not protected by a mutex, it is possible to reenter the function during external calls, such as that to `doTransferIn`. If Minterest ever allows additional liquidators to join the protocol (and thus to call `autoLiquidationRepayDeadBorrow`), it should further scrutinize the effect of this reentrancy on the contract's bookkeeping (TOB-MNTR-13).

```
actualRepayAmount = Math.min(_accountBorrows, _totalProtocolInterest);
uint256 totalBorrowsNew = totalBorrows.sub(actualRepayAmount);
uint256 accountBorrowsNew = _accountBorrows.sub(actualRepayAmount);
uint256 totalProtocolInterestNew = _totalProtocolInterest.sub(actualRepayAmount);
totalBorrows = totalBorrowsNew;
accountBorrows[borrower].principal = accountBorrowsNew;
accountBorrows[borrower].interestIndex = borrowIndex;
totalProtocolInterest = totalProtocolInterestNew;
```

Figure 17.1: Part of the `autoLiquidationRepayDeadBorrow` function

Recommendations

Short term, implement safeguards for privileged actions such as calls to `autoLiquidationRepayDeadBorrow` and develop documentation on the use of those actions. Minterest should also consider adding a `nonReentrant` modifier to `autoLiquidationRepayDeadBorrow` if it plans to add third-party liquidators to the protocol.

Long term, refactor the accounting code to follow the checks-effects-interactions pattern, and enforce restrictions on when admins can perform privileged actions.

Summary of Recommendations

The Minterest smart contracts are a work in progress with multiple planned iterations. Trail of Bits recommends that Minterest Finance address the findings detailed in this report and take the following additional steps prior to deployment:

- Reduce the use of long and nested loops as well as gas-intensive operations such as the allocation of large arrays and sorting. Because Minterest cites its unique liquidation scheme as its main advantage over its competitors, it is critical to ensure that liquidation-related code paths are as efficient as possible to maximize returns.
- Increase the readability of the code by updating its comments, maintaining a consistent style, removing unused code and variables, and making function names more accurate.
- Thoroughly review and document the current proxy architecture and its pitfalls, and carefully consider whether it is necessary to use proxies.
- Change the liquidation model to account for the risk of front- and back-running during liquidation transactions.

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

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. Estimated Gas Costs Per Method

We generated the following estimates by using `hardhat-gas-reporter` alongside Minterest's existing test suite.

Method	Min	Max	Avg.	# Calls
`AccessControlMock.forAdmin`	23509	28539	26024	2
`AccessControlMock.forAdminOr`	24002	29063	26533	4
`AccessControlMock.forAnyone`	21272	26302	23787	6
`AccessControlMock.setAdminPub`	26944	44044	31219	4
`BoolSupervisor.setAutoLiquidationRepayDeadBorrowAllowed`	-	-	26555	1
`BoolSupervisor.setAutoLiquidationSeizeAllowed`	-	-	26577	1
`BoolSupervisor.setBorrowAllowed`	23745	26533	23931	15
`BoolSupervisor.setBorrowVerify`	-	-	23723	14
`BoolSupervisor.setLendAllowed`	23732	26520	23918	15
`BoolSupervisor.setLendVerify`	-	-	23778	14
`BoolSupervisor.setRedeemAllowed`	23745	26533	23952	27
`BoolSupervisor.setRedeemVerify`	-	-	23729	25
`BoolSupervisor.setRepayBorrowAllowed`	23810	26598	23996	30
`BoolSupervisor.setRepayBorrowVerify`	-	-	23800	28
`BoolSupervisor.setTransferAllowed`	-	-	26533	1
`Buyback.buyback`	64648	98860	96005	12
`Buyback.leave`	36510	82879	63437	58
`Buyback.participate`	77498	123471	79083	29
`Buyback.restake`	56471	139341	85551	14

Buyback.restakeFor	-	-	100642	1
Buyback.stake	105048	190572	166007	59
Buyback.unstake	76806	99135	92751	7
BuybackDripper.drip	41806	132629	79527	53
BuybackDripper.setPeriodDuration	-	-	29747	3
BuybackDripper.setPeriodRate	29684	46844	42955	22
ChainlinkFeedMock.reportNewRound	96312	116584	115554	28
ChainlinkPriceOracle.setTimestampThreshold	-	-	30054	2
ChainlinkPriceOracle.setTokenConfig	91576	91648	91621	20
DefectiveToken.allocateTo	-	-	68315	2
DefectiveToken.approve	-	-	46189	2
DefectiveToken.setFail	21740	43652	32696	2
ERC20Harness.approve	26230	46490	42444	116
ERC20Harness.harnessSetBalance	22230	44214	38893	237
ERC20Harness.harnessSetFailTransferFromAddress	24297	44221	25125	73
ERC20Harness.harnessSetFailTransferToAddress	24274	44186	25749	27
ERC20Harness.transfer	-	-	53806	7
FaucetNonStandardToken.allocateTo	34056	68364	52851	199
FaucetNonStandardToken.approve	46140	46236	46187	32
FaucetToken.allocateTo	34067	68351	60044	244
FaucetToken.approve	26288	46488	45895	202
FaucetTokenReEntrantHarness.allocateTo	-	-	68315	1
FaucetTokenReEntrantHarness.approve	-	-	49875	1
FeeToken.allocateTo	34103	68303	60973	7

`FeeToken.approve`	-	-	46176	5
`FeeToken.transfer`	-	-	73869	1
`InterestRateModelHarness.setBorrowRate`	21670	43642	28052	31
`InterestRateModelHarness.setFailBorrowRate`	23719	43631	25597	106
`Liquidation._setPriceOracle`	32758	52768	50280	61
`LiquidationMock._addTrustedLiquidator`	-	-	47610	2
`LiquidationMock._removeTrustedLiquidator`	-	-	25777	2
`LiquidationMock._setPartialLiquidationMaxAttempts`	-	-	30177	4
`LiquidationMock._setPartialLiquidationMinSum`	30235	47383	33689	5
`LiquidationMock._setPriceOracle`	-	-	40525	2
`LiquidationMock._setSafeDeviationThreshold`	-	-	30314	2
`LiquidationMock._setSafetyBuffer`	30216	30228	30222	4
`LiquidationMock._setSupervisor`	-	-	36316	2
`LiquidationMock._setSwapRouterPlusFee`	-	-	37193	1
`LiquidationMock._setTreasury`	-	-	30535	2
`LiquidationMock.doSwapPub`	-	-	596610	1
`LiquidationMock.forceSetSupervisor`	-	-	26940	1
`LiquidationMock.harnessSetAccountState`	125078	272590	180918	3
`LiquidationMock.liquidateForgivableLoan`	332512	1356937	1013858	6
`LiquidationMock.liquidateUnsafeLoan`	1075622	1551884	1352637	40
`LiquidationMock.mutateAccountLiquidationAttemptsPub`	22428	44406	32195	5
`LiquidationMock.repayDeadBorrowsPub`	-	-	76876	4
`LiquidationMock.repayPub`	118365	150581	136135	11
`LiquidationMock.resetAccountLiquidationAttempts`	34776	35164	34970	2

`LiquidationMock.seizePub`	-	-	118197	3
`LiquidationMock.setAccountAttemptsMock`	-	-	44147	1
`LiquidationMock.setAccountMaxLiquidationAttempts`	46068	46080	46074	2
`LiquidationMock.transferSurplusPub`	-	-	55762	1
`MinterestNFT.mint`	157123	174223	165673	4
`MinterestNFT.setSupervisor`	-	-	56124	2
`MinterestNFT.setWhitelist`	-	-	51152	2
`MinterestNFTMock.createTiers`	79555	744625	281059	58
`MinterestNFTMock.enableEmissionBoosting`	-	-	169075	7
`MinterestNFTMock.enableTiers`	188030	688771	293516	55
`MinterestNFTMock.harnessDisableEmissionBoosting`	-	-	21527	1
`MinterestNFTMock.harnessEnableEmissionBoosting`	-	-	43410	11
`MinterestNFTMock.harnessSetBlockNumber`	23719	43619	28477	243
`MinterestNFTMock.harnessSetBorrowEmissionBoost`	-	-	66644	2
`MinterestNFTMock.harnessSetMarketBorrowSpecificData`	-	-	67158	1
`MinterestNFTMock.harnessSetMarketIndexes`	26998	66810	66324	82
`MinterestNFTMock.harnessSetMarketSupplySpecificData`	67094	67214	67208	45
`MinterestNFTMock.harnessSetSupplyEmissionBoost`	-	-	66620	2
`MinterestNFTMock.mint`	162239	174168	171186	4
`MinterestNFTMock.mintBatch`	167161	2006662	953767	36
`MinterestNFTMock.safeBatchTransfer`	187836	319073	276258	9
`MinterestNFTMock.safeTransfer`	177727	260227	232373	6
`MinterestNFTMock.setMinter`	-	-	47667	4
`MinterestNFTMock.setSupervisor`	-	-	56058	47

MinterestNFTMock.setURI	-	-	34363	2
MinterestNFTMock.setWhitelist	-	-	51173	34
MinterestNFTMock.updateBorrowIndexesHistory	27040	205937	46661	70
MinterestNFTMock.updateSupplyIndexesHistory	26996	206052	46629	70
MinterestNFTScenario.createTiers	-	-	628767	5
MinterestNFTScenario.mintBatch	-	-	12087250	5
MinterestNFTScenario.safeTransfer	408640	4514772	2570725	14
MinterestNFTScenario.setSupervisor	-	-	56124	1
MinterestNFTScenario.setWhitelist	-	-	51152	1
MintProxy.changeAdmin	-	-	28650	5
MintProxy.upgradeTo	-	-	32982	1
Mnt.approve	29186	46286	34404	23
Mnt.delegate	-	-	95244	2
Mnt.delegateBySig	-	-	76548	1
Mnt.transfer	39011	102936	55521	64
MntScenario.approve	46262	46574	46304	8
MntScenario.delegate	48342	110571	86920	59
MntScenario.transfer	29339	56135	48987	28
MntScenario.transferFrom	46816	63916	60496	5
MntScenario.transferFromScenario	-	-	77276	3
MntScenario.transferScenario	-	-	135535	3
MNTSource.drip	48419	91029	72527	6
MToken._addProtocolInterest	73142	132042	89025	13
MToken._reduceProtocolInterest	81153	90753	87153	4

`MToken._setProtocolInterestFactor`	49623	110571	77921	29
`MToken._setWhitelist`	-	-	56210	3
`MToken.accrueInterest`	70246	97050	91192	32
`MToken.borrow`	166209	1305837	273640	102
`MToken.lend`	120391	1264363	191762	194
`MToken.redeem`	107313	207314	159209	20
`MToken.redeemUnderlying`	111430	196705	167597	15
`MToken.repayBorrow`	94719	212540	146592	48
`MToken.repayBorrowBehalf`	147541	150515	149028	2
`MToken.sweepToken`	61245	61532	61389	2
`MTokenHarness._delegateMntLikeTo`	-	-	56021	1
`MTokenHarness._reduceProtocolInterest`	57821	104342	69451	4
`MTokenHarness._setInterestRateModel`	-	-	38648	2
`MTokenHarness._setProtocolInterestFactor`	32136	102325	54528	10
`MTokenHarness._setSupervisor`	-	-	34129	2
`MTokenHarness._setWhitelist`	-	-	51228	8
`MTokenHarness.accrueInterest`	25603	67141	42218	5
`MTokenHarness.autoLiquidationRepayDeadBorrow`	31544	47322	40458	6
`MTokenHarness.autoLiquidationSeize`	-	-	91486	2
`MTokenHarness.borrow`	168808	242949	192565	6
`MTokenHarness.harnessCallBorrowAllowed`	94495	157417	136443	3
`MTokenHarness.harnessCallResetLiquidationAttempts`	39267	55446	44664	3
`MTokenHarness.harnessExchangeRateDetails`	26455	88395	55898	13
`MTokenHarness.harnessFastForward`	26590	48986	38234	72

MTokenHarness.harnessIncrementTotalBorrows	26656	43756	28794	8
MTokenHarness.harnessLendFresh	-	-	126997	5
MTokenHarness.harnessRedeemFresh	83909	83936	83923	8
MTokenHarness.harnessRepayBorrowFresh	83013	83023	83018	8
MTokenHarness.harnessSetAccountBorrows	24711	66967	58315	105
MTokenHarness.harnessSetAccrualBlockNumber	23752	43652	42928	55
MTokenHarness.harnessSetBalance	22301	44297	39158	83
MTokenHarness.harnessSetBlockNumber	23753	43917	42941	55
MTokenHarness.harnessSetBorrowIndex	23744	26856	24688	85
MTokenHarness.harnessSetExchangeRate	23983	65891	61191	76
MTokenHarness.harnessSetFailTransferToAddress	24398	44310	26886	16
MTokenHarness.harnessSetInterestRateModelFresh	-	-	38409	2
MTokenHarness.harnessSetTotalBorrows	23715	43999	39325	93
MTokenHarness.harnessSetTotalProtocolInterest	43606	43978	43680	15
MTokenHarness.harnessSetTotalSupply	23697	43657	42197	70
MTokenHarness.lend	125131	204880	147930	39
MTokenHarness.redeem	109473	129476	119475	2
MTokenHarness.redeemUnderlying	96477	129539	113008	2
MTokenHarness.repayBorrow	93978	108378	103656	4
MTokenHarness.repayBorrowBehalf	-	-	106459	1
MTokenHarness.transfer	94756	136156	115590	7
MTokenImmutable.lend	-	-	207597	3
MTokenScenario.setTotalBorrows	48729	48765	48747	2
MTokenScenario.setTotalProtocolInterest	48640	48676	48658	2

`PriceOracleProxy.setSaiPrice`	45825	45849	45837	2
`SimplePriceOracle.setDirectPrice`	45861	45933	45890	7
`SimplePriceOracle.setUnderlyingPrice`	24153	46173	41897	106
`Supervisor._grantMnt`	50100	71722	63007	6
`Supervisor._setBorrowPaused`	36665	58710	49237	7
`Supervisor._setBuyback`	-	-	52703	1
`Supervisor._setGateKeeper`	32870	52770	47338	22
`Supervisor._setLendPaused`	36675	58699	51324	6
`Supervisor._setLiquidationFee`	35640	55540	55227	72
`Supervisor._setLiquidator`	52681	52703	52700	7
`Supervisor._setMarketBorrowCaps`	56934	81793	61915	5
`Supervisor._setMinterestNFT`	56285	56330	56326	12
`Supervisor._setMntEmissionRate`	53625	103297	94934	59
`Supervisor._setTransferPaused`	36392	36493	36437	7
`Supervisor._setUtilizationFactor`	35768	85266	65029	106
`Supervisor._supportMarket`	90774	110146	100900	186
`Supervisor.accrueAutoLiquidation`	89271	357536	345872	23
`Supervisor.allowWithdraw`	-	-	50772	52
`Supervisor.autoLiquidationSeizeAllowed`	-	-	99715	2
`Supervisor.denyWithdraw`	-	-	28851	3
`Supervisor.disableAsCollateral`	46359	186191	107600	27
`Supervisor.enableAsCollateral`	32408	295591	113786	151
`Supervisor.prepareNftTransfer`	-	-	401668	4
`Supervisor.updateAndGetMntIndexes`	-	-	56477	1

Supervisor.withdrawMnt	114793	143810	134138	3
Supervisor.withdrawMnt	113119	1515067	535158	19
Supervisor.withdrawMnt	636080	858564	753776	3
SupervisorHarness.harnessDistributeAllSupplierMnt	91937	109171	103426	3
SupervisorHarness.harnessDistributeBorrowerMnt	43765	75656	64716	4
SupervisorHarness.harnessDistributeSupplierMnt	-	-	43277	1
SupervisorHarness.harnessTransferMnt	27341	68266	43594	3
SupervisorHarness.harnessUpdateMntBorrowIndex	42264	77555	54292	8
SupervisorHarness.harnessUpdateMntSupplyIndex	41654	76901	54142	8
SupervisorHarness.setBlockNumber	28704	48616	44821	15
SupervisorHarness.setMntAccrued	49217	49277	49232	4
SupervisorHarness.setMntAddress	29110	49010	47688	56
SupervisorHarness.setMntBorrowerState	30281	50313	46304	5
SupervisorHarness.setMntBorrowState	-	-	49712	3
SupervisorHarness.setMntSupplierState	50291	50303	50295	3
SupervisorHarness.setMntSupplyState	-	-	49667	4
SupervisorHarness.unlist	-	-	27185	3
SupervisorScenario.fastForward	31883	49007	44607	148
UniswapV2Router02.addLiquidity	2190360	2214618	2196549	24
UniswapV2Router02.swapExactTokensForTokens	-	-	144172	1
Vesting.createVestingSchedule	89878	114781	104820	10
Vesting.revokeVestingSchedule	73605	97031	79373	9
Vesting.setBuyback	-	-	47611	1
Vesting.withdraw	53064	105923	75518	8

`WBTCToken.allocateTo`	34143	68391	60657	60
`WBTCToken.approve`	48412	48460	48422	68
`WBTCToken.pause`	-	-	27439	8
`Whitelist.addMember`	54926	72038	65733	19
`Whitelist.removeMember`	-	-	28385	2
`Whitelist.setMaxMembers`	-	-	32067	2
`Whitelist.turnOffWhitelistMode`	73916	2324455	856503	9

D. Code Quality Recommendations

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

Correctness and Style

- Avoid using `pragma experimental ABIEncoderV2`, as the encoder is enabled by default in Solidity 0.8.0 and is no longer experimental. (It can still be declared explicitly using `pragma ABIEncoderV2`.)

- Use modifiers consistently throughout the codebase. Currently, some contracts make extensive use of modifiers, while others perform checks at the function level.

- Maintain a consistent code style across the codebase. For example, use the underscore character consistently. Currently, `WhitelistInterface.sol` uses an underscore for function parameters, while `SupervisorInterface.sol` uses one only in some function parameters. `MTokenInterface.sol` uses the character for function names, not for parameters.

- Review unfinished functions such as the following:

    - `updateMntStateIfNeeded` in `MinterestNFT.sol`, which has an unused parameter and a TODO comment

    - `_beforeTokenTransfer` in `MinterestNFT.sol`, which has a TODO comment on one code branch

- Use a leading underscore to denote internal functions (i.e., `_f` rather than `fInternal`). These functions include `enableMarketAsCollateralInternal()` (`Supervisor.sol:138`).

- Have functions explicitly revert upon an error, rather than returning and assuming the call will revert later.

    - For example, when the `restakeFor()` function (`Buyback.sol:189`) is called on an account that is not participating in the buyback being performed, it returns rather than reverting. In other functions in the same contract, this same case generally causes a revert.

- Ensure that functions' names reflect their actual purposes.

    - For example, the `accrueAutoLiquidation()` function (`Supervisor.sol:384-390`) has little to do with the liquidation process and does not execute any actions automatically; consider renaming it `accrueBorrowersInterest()`.

- The `maxMembers` variable in `Whitelist.sol` is defined as a `uint256`, while both the `constructor` and the `setMaxMembers()` function define it as a `uint8`. It is unclear whether this is an intentional discrepancy meant to implement a technical limit. If it is, we would recommend improving the related documentation.

- Avoid type confusion and coercion. Use address types instead of contract types where possible.

  - For example, `oracle.getUnderlyingPrice(MToken(address(stable)))`, at `Liquidation.sol:L680-L682`, coerces an ERC20 to an MToken even though it may not be an MToken. Consider refactoring `getUnderlyingPrice` to use `address` instead of `MToken` as the type.

Optimization

- Avoid mixing `SafeMath` and Solidity 0.8.0's checked math in the codebase.

  - Some contracts use either `SafeMath` or checked math, while others, like `Vesting.sol`, use both.

  - Simplify the code and reduce gas costs by using Solidity 0.8.0's checked math instead of `SafeMath` in `require()` statements such as that in `drip()` (`BuybackDripper.sol:L71-L72`).

- Consider refactoring `MTokenInterface.sol` to avoid inheriting from `MTokenStorage.sol.`

  - The current implementation uses an unusual architecture in which contracts that interact with `MToken.sol` import the entire contract instead of just its interface.

- Remove unused variables (e.g., `mntRate` in `SupervisorV1Storage.sol`).

- Reduce the storage costs by merging redundant mappings.

  - Consider making `isParticipating` a field of the `Member` struct and using a single mapping to associate the `discounted` and `staked` values of an address (`Buyback.sol:66-79`).

- Reduce the gas costs by removing redundant checks and running checks that are simple or often fail as early as possible.

- Move the `accountMembership` check at `Supervisor.sol:176-179` to the beginning of `disableAsCollateral()`.
- De-duplicate the checks of `drRateMantissa == expScale` and `approveDrRate()` (`Liquidation.sol#L270-L320`).
- De-duplicate the checks for tier activity in `update{Borrow,Supply}IndexesHistory()` at `MinterestNFT.sol:690`.

- To save gas, avoid traversing and sorting large arrays whenever possible, and use an efficient method when those operations are unavoidable.
  - Refactor the $O(n^2)$-complexity function `sortPoints()` (`MinterestNFT.sol:560`). Consider computing the delta of each point when it is placed in the array or sorting the array inside the loop in `createBoostSegments()`.

- Optimize the `withdraw()` function in `Vesting.sol` by having it send as large an amount as possible instead of reverting when the `amount` of a withdrawal exceeds the contract's `balance`.
  - In the current implementation, if a prior withdrawal has left the contract with insufficient funds to execute a withdrawal subsequently requested by a different user, that user's call to `withdraw(amount)` will revert, but the user will still incur a gas cost.

E. Token Integration Checklist

The following checklist provides recommendations for interactions with arbitrary tokens. Every unchecked item should be justified, and its associated risks, understood. An up-to-date version of the checklist can be found in `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
```

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 Security 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.

ERC Conformity

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.

- ❏ **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.

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.

Finally, there are certain characteristics that are difficult to identify automatically. Conduct a manual review of the following conditions:

- ❏ **`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.

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 if 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.

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.

F. Slither Script for Detecting Reentrancies

This script can be used to verify that the functions that write to `accountBorrows` cannot be reentered, which could result in a loss of funds (TOB-MNTR-13). The `autoLiquidationRepayDeadBorrow` function is whitelisted because it can be called only by liquidators who are trusted.

```
from typing import List
from slither import Slither
from slither.core.declarations import Contract
from slither.core.declarations import Modifier
slither = Slither(".", ignore_compile=True)
def find_contract_in_compilation_units(contract_name: str) -> Contract:
    contracts = slither.get_contract_from_name(contract_name)
    return (contracts[0]) if len(contracts)>0 else print("Contract not found")

def _check_reentrant_destructive_write(
    contract: Contract, modifier: str, protected_variable: str, whitelisted_functions:
List[str]
):
    print(f"### Verify functions that write to {protected_variable} call {modifier}
modifier")
    no_bug_found = True
    for function in contract.functions:
        if function.is_constructor or function.view or function.name in
whitelisted_functions:
            continue

        internal_modifiers = [candidate for candidate in function.all_internal_calls() if
isinstance(candidate, Modifier)]
        for x in function.all_state_variables_written():
            if x.name == protected_variable:
                if not function.modifiers or (
                    not any((str(x) == modifier) for x in function.modifiers +
internal_modifiers)
                ):
                    print(f"\t- {function.canonical_name} should have a {modifier}
modifier")
                    no_bug_found = False

    if no_bug_found:
        print("\t- No bug found")


_check_reentrant_destructive_write(
    find_contract_in_compilation_units("MToken"),
    "nonReentrant",
    "accountBorrows",
    ["autoLiquidationRepayDeadBorrow"]
)
```

G. Liquidation Scheme Design

Minterest's reliance on an external decentralized exchange and permissioned actors exposes the protocol to denial-of-service and sandwich attack risks (TOB-MNTR-2). Such attacks could prevent its bot operators from performing swaps without excessive slippage, increasing the risk of under-collateralization. While the design seeks to maximize the amount of interest paid out to liquidity providers, that is not the most important aspect of a liquidation system.

To prevent the accumulation of insolvent loans, we recommend increasing the number of actors that can perform liquidations and encouraging competition; these steps will also ensure that liquidations are completed quickly and efficiently, which will benefit both the protocol and its users. Additional details on the purpose and trade-offs of liquidation incentives are provided below.

In a lending protocol, the primary goal of a liquidation scheme is avoiding under-collateralization. Thus, when the value of a user's debt exceeds that of the user's collateral, the protocol will incentivize bot operators to liquidate the position by offering them a "bonus" (a discount on the collateral) in exchange for removing it from the system. If the liquidation is not executed in a timely manner, the protocol will continue to accumulate bad debt; eventually, liquidity providers will be unable to withdraw the assets they have provided, because the protocol will have more liabilities than assets. Therefore, when a loan has become under-collateralized, it is crucial that it be liquidated as soon as possible, and for a low cost. In other words, a protocol should allow users to keep as much collateral as possible by paying bot operators the lowest amount possible to perform liquidations.

Rather than offering a fixed bonus to liquidators like Aave and Compound do, some protocols increase the amount of the bonus over time. Lending protocols such as MakerDAO and Euler Finance use Dutch auctions to increase the amount of the discount on collateral; the debt auctions start at a fixed price, and the price of the debt decreases. This model requires bots to accept the least profitable bonus rate, making them more efficient; this means that if one bot can perform a liquidation at less of a discount than another, the user will be left with more collateral than a fixed-bonus system would provide. Note, though, that MakerDAO and Euler Finance have other idiosyncrasies not mentioned here that should be evaluated as Minterest Finance considers the design of its liquidation system.

References

- Euler Finance Liquidations
- MakerDAO Liquidations
- Gauntlet's Risk Assessment of Compound

H. Fix Log

On March 7, 2022, Trail of Bits reviewed the fixes and mitigations implemented by the Minterest Finance team for the issues identified in this report. The team fixed nine of the issues, accepted the risks associated with three, identifying those issues as intended behavior, and left three unaddressed; the last two could not be definitively verified as fixed or not fixed. We reviewed each of the fixes to ensure that the proposed remediation would be effective. For additional information, please refer to the Detailed Fix Log below.

ID	Title	Severity	Fix Status
1	MinterestNFT batched transfers can cause bookkeeping errors	High	Fixed
2	Risk of a systemic liquidation failure due to a denial of service	High	Undetermined
3	Excessive gas costs due to the repeated allocation of a large array	Medium	Fixed
4	Risks associated with using MintProxy alongside contracts that implement AccessControl	Low	Fixed
5	Error-prone empty function	Low	Fixed
6	Inconsistent definition of total supply	Low	Fixed
7	Input validation of Chainlink oracle	Low	Fixed
8	setAdmin does not emit events	Low	Fixed
9	Block-time assumption may cause interest-calculation discrepancies	Low	Not Fixed
10	Market-addition costs increase linearly	Informational	Fixed
11	Incorrect check of whether a contract is an ERC20	Informational	Fixed
12	Block number overflow	Informational	Not Fixed

13	Potential reentrancy vulnerability in repayBorrow	Informational	Risk Accepted
14	Users can borrow assets they are actively using as collateral	Undetermined	Risk Accepted
15	Unbounded loop could enable a denial of service	Undetermined	Not Fixed
16	Stable parameter is not guaranteed to be a stablecoin	Undetermined	Undetermined
17	autoLiquidationRepayDeadBorrow's lack of data validation	Undetermined	Risk Accepted

Detailed Fix Log

TOB-MNTR-1: MinterestNFT batched transfers can cause bookkeeping errors
Fixed in `b0280e96`. The problematic `return` statement was changed to a `continue` statement.

TOB-MNTR-2: Risk of a systemic liquidation failure due to a denial of service
Undetermined. The client claims that this issue was addressed through commits `32066e1e`, `6ace6c28`, `ea2cbc01`, `0d3aba85`, and `4a8e7a7b`; however, because of the complexity of the changes, it was impossible to verify the status of the fix in the short time allotted for the fix review. We suggest that the Minterest Finance team perform simulations to determine whether the assumptions present in the new code are valid.

TOB-MNTR-3: Excessive gas costs due to the repeated allocation of a large array
Fixed in PR #69. The large array allocation was removed.

TOB-MNTR-4: Risks associated with using MintProxy alongside contracts that implement AccessControl
Fixed in PR #80. The problematic code was removed entirely.

TOB-MNTR-5: Error-prone empty function
Fixed in `98d6d07b`. The empty function now reverts when called.

TOB-MNTR-6: Inconsistent definition of total supply
Fixed in `df3b1a60`. The outdated comment was corrected.

TOB-MNTR-7: Input validation of Chainlink oracle
Fixed in `284b94e5`. The unnecessary `require` statement was removed, and a new one was added to ensure that `answeredInRound == roundId`.

TOB-MNTR-8: setAdmin does not emit events
Fixed in `a5abcabe`. The function `setAdmin()` now emits an event, `NewAdminSet`.

TOB-MNTR-9: Block-time assumption may cause interest-calculation discrepancies
Not fixed. The Minterest Finance team has not addressed or commented on the issue.

TOB-MNTR-10: Market-addition costs increase linearly
Fixed in `2f20b887`. The unnecessary `for` loop was removed.

TOB-MNTR-11: Incorrect check of whether a contract is an ERC20
Fixed in `c9dd4a82`. The unnecessary call to `totalSupply()` was removed.

TOB-MNTR-12: Block number overflow

Not fixed. The Minterest Finance team has not addressed or commented on the issue.

TOB-MNTR-13: Potential reentrancy vulnerability in repayBorrow
Risk accepted. The client acknowledged the reentrancy risk that would occur if the `nonReentrant` modifier were removed.

TOB-MNTR-14: Users can borrow assets they are actively using as collateral
Risk accepted. The client considers this issue to be expected behavior.

TOB-MNTR-15: Unbounded loop could enable a denial of service
Not fixed. The Minterest Finance team has not addressed or commented on the issue.

TOB-MNTR-16: Stable parameter is not guaranteed to be a stablecoin
Undetermined. The client claims that the code in question will be reworked as part of a larger overhaul of Minterest's liquidation functionality; however, Trail of Bits was unable to definitively verify that claim.

TOB-MNTR-17: autoLiquidationRepayDeadBorrow's lack of data validation
Risk accepted. The client does not plan to allow third parties to perform liquidations; this approach will prevent third-party exploitation. However, the client has not added any restrictions on the loans or the amount of debt that can be removed from the system by an administrator.