SMART CONTRACT AUDIT REPORT

for

Minterest Protocol

Prepared By: Xiaomi Huang

PeckShield

June 28, 2022

Document Properties

Client	Minterest
Title	Smart Contract Audit Report
Target	Minterest
Version	1.0
Author	Xuxian Jiang
Auditors	Shulin Bie, Xuxian Jiang
Reviewed by	Xiaomi Huang
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	June 28, 2022	Xuxian Jiang	Final Release
1.0-rc1	May 28, 2022	Xuxian Jiang	Release Candidate #1

Contact

For more information about this document and its contents, please contact PeckShield Inc.

Name	Xiaomi Huang
Phone	+86 183 5897 7782
Email	contact@peckshield.com

Contents

1 Introduction

Given the opportunity to review the design document and related smart contract source code of the Minterest protocol, we outline in the report our systematic approach to evaluate potential security issues in the smart contract implementation, expose possible semantic inconsistencies between smart contract code and design document, and provide additional suggestions or recommendations for improvement. Our results show that the given version of smart contracts can be further improved due to the presence of several issues related to either security or performance. This document outlines our audit results.

1.1 About Minterest

Minterest is a decentralised lending protocol. It comes with a unique economic model where the protocol itself captures 100% of the value created from its functions, including interest, flash loan, and auto-liquidation fees. This value is exchanged via an automated Buyback process for its native MNT token which is then distributed to its users in return for their participation in the protocol's governance. The basic information of the audited protocol is as follows:

Table 1.1: Basic Information of Minterest

Item	Description
Target	Minterest
Website	https://minterest.com/
Type	EVM Smart Contract
Language	Solidity
Audit Method	Whitebox
Latest Audit Report	June 28, 2022

In the following, we show the Git repository of reviewed files and the commit hash value used in this audit.

- https://github.com/minterest-finance/protocol.git (16a4862)

And here is the commit IDs after all fixes for the issues found in the audit have been checked in:

- https://github.com/minterest-finance/protocol.git (8fb46e6)

1.2 About PeckShield

PeckShield Inc. [12] is a leading blockchain security company with the goal of elevating the security, privacy, and usability of current blockchain ecosystems by offering top-notch, industry-leading services and products (including the service of smart contract auditing). We are reachable at Telegram (https://t.me/peckshield), Twitter (http://twitter.com/peckshield), or Email (contact@peckshield.com).

Table 1.2: Vulnerability Severity Classification

Impact \ Likelihood	*High*	*Medium*	*Low*
High	Critical	High	Medium
Medium	High	Medium	Low
Low	Medium	Low	Low

1.3 Methodology

To standardize the evaluation, we define the following terminology based on OWASP Risk Rating Methodology [11]:

- Likelihood represents how likely a particular vulnerability is to be uncovered and exploited in the wild;
- Impact measures the technical loss and business damage of a successful attack;
- Severity demonstrates the overall criticality of the risk.

Likelihood and impact are categorized into three ratings: *H*, *M* and *L*, i.e., *high*, *medium* and *low* respectively. Severity is determined by likelihood and impact and can be classified into four categories accordingly, i.e., *Critical*, *High*, *Medium*, *Low* shown in Table 1.2.

To evaluate the risk, we go through a list of check items and each would be labeled with a severity category. For one check item, if our tool or analysis does not identify any issue, the

Table 1.3: The Full List of Check Items

Category	Check Item
Basic Coding Bugs	Constructor Mismatch
	Ownership Takeover
	Redundant Fallback Function
	Overflows & Underflows
	Reentrancy
	Money-Giving Bug
	Blackhole
	Unauthorized Self-Destruct
	Revert DoS
	Unchecked External Call
	Gasless Send
	Send Instead Of Transfer
	Costly Loop
	(Unsafe) Use Of Untrusted Libraries
	(Unsafe) Use Of Predictable Variables
	Transaction Ordering Dependence
	Deprecated Uses
Semantic Consistency Checks	Semantic Consistency Checks
Advanced DeFi Scrutiny	Business Logics Review
	Functionality Checks
	Authentication Management
	Access Control & Authorization
	Oracle Security
	Digital Asset Escrow
	Kill-Switch Mechanism
	Operation Trails & Event Generation
	ERC20 Idiosyncrasies Handling
	Frontend-Contract Integration
	Deployment Consistency
	Holistic Risk Management
Additional Recommendations	Avoiding Use of Variadic Byte Array
	Using Fixed Compiler Version
	Making Visibility Level Explicit
	Making Type Inference Explicit
	Adhering To Function Declaration Strictly
	Following Other Best Practices

contract is considered safe regarding the check item. For any discovered issue, we might further deploy contracts on our private testnet and run tests to confirm the findings. If necessary, we would additionally build a PoC to demonstrate the possibility of exploitation. The concrete list of check items is shown in Table 1.3.

In particular, we perform the audit according to the following procedure:

- Basic Coding Bugs: We first statically analyze given smart contracts with our proprietary static code analyzer for known coding bugs, and then manually verify (reject or confirm) all the issues found by our tool.
- Semantic Consistency Checks: We then manually check the logic of implemented smart contracts and compare with the description in the white paper.
- Advanced DeFi Scrutiny: We further review business logics, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.
- Additional Recommendations: We also provide additional suggestions regarding the coding and development of smart contracts from the perspective of proven programming practices.

To better describe each issue we identified, we categorize the findings with Common Weakness Enumeration (CWE-699) [10], which is a community-developed list of software weakness types to better delineate and organize weaknesses around concepts frequently encountered in software development. Though some categories used in CWE-699 may not be relevant in smart contracts, we use the CWE categories in Table 1.4 to classify our findings.

1.4 Disclaimer

Note that this security audit is not designed to replace functional tests required before any software release, and does not give any warranties on finding all possible security issues of the given smart contract(s) or blockchain software, i.e., the evaluation result does not guarantee the nonexistence of any further findings of security issues. As one audit-based assessment cannot be considered comprehensive, we always recommend proceeding with several independent audits and a public bug bounty program to ensure the security of smart contract(s). Last but not least, this security audit should not be used as investment advice.

Table 1.4: Common Weakness Enumeration (CWE) Classifications Used in This Audit

Category	Summary
Configuration	Weaknesses in this category are typically introduced during the configuration of the software.
Data Processing Issues	Weaknesses in this category are typically found in functionality that processes data.
Numeric Errors	Weaknesses in this category are related to improper calculation or conversion of numbers.
Security Features	Weaknesses in this category are concerned with topics like authentication, access control, confidentiality, cryptography, and privilege management. (Software security is not security software.)
Time and State	Weaknesses in this category are related to the improper management of time and state in an environment that supports simultaneous or near-simultaneous computation by multiple systems, processes, or threads.
Error Conditions, Return Values, Status Codes	Weaknesses in this category include weaknesses that occur if a function does not generate the correct return/status code, or if the application does not handle all possible return/status codes that could be generated by a function.
Resource Management	Weaknesses in this category are related to improper management of system resources.
Behavioral Issues	Weaknesses in this category are related to unexpected behaviors from code that an application uses.
Business Logics	Weaknesses in this category identify some of the underlying problems that commonly allow attackers to manipulate the business logic of an application. Errors in business logic can be devastating to an entire application.
Initialization and Cleanup	Weaknesses in this category occur in behaviors that are used for initialization and breakdown.
Arguments and Parameters	Weaknesses in this category are related to improper use of arguments or parameters within function calls.
Expression Issues	Weaknesses in this category are related to incorrectly written expressions within code.
Coding Practices	Weaknesses in this category are related to coding practices that are deemed unsafe and increase the chances that an exploitable vulnerability will be present in the application. They may not directly introduce a vulnerability, but indicate the product has not been carefully developed or maintained.

2 | Findings

2.1 Summary

Here is a summary of our findings after analyzing the implementation of the `Minterest` protocol. During the first phase of our audit, we study the smart contract source code and run our in-house static code analyzer through the codebase. The purpose here is to statically identify known coding bugs, and then manually verify (reject or confirm) issues reported by our tool. We further manually review business logic, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.

Severity	# of Findings	
Critical	0	
High	0	
Medium	2	■■
Low	5	■■■■■
Informational	0	
Total	7	

We have so far identified a list of potential issues: some of them involve subtle corner cases that might not be previously thought of, while others refer to unusual interactions among multiple contracts. For each uncovered issue, we have therefore developed test cases for reasoning, reproduction, and/or verification. After further analysis and internal discussion, we determined a few issues of varying severities need to be brought up and paid more attention to, which are categorized in the above table. More information can be found in the next subsection, and the detailed discussions of each of them are in Section 3.

2.2 Key Findings

Overall, these smart contracts are well-designed and engineered, though the implementation can be improved by resolving the identified issues (shown in Table 2.1), including 2 medium-severity vulnerabilities and 5 low-severity vulnerabilities.

Table 2.1: Key Minterest Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Improved Allowance Management in DeadDrop	Coding Practices	Resolved
PVE-002	Low	Improved ERC20-Compliance Of MToken	Coding Practices	Resolved
PVE-003	Low	Possible Front-Running For Unintended Payment In repayBorrowBehalf()	Time and State	Resolved
PVE-004	Medium	Possible Sandwich/MEV Attacks For Drip Collection	Time and State	Resolved
PVE-005	Medium	Trust Issue of Admin Keys	Security Features	Mitigated
PVE-006	Low	Accommodation of Non-ERC20-Compliant Tokens	Business Logic	Resolved
PVE-007	Low	Improved Gas Efficiency in Emission-Booster::enableTiersInternal()	Coding Practices	Resolved

Besides the identified issues, we emphasize that for any user-facing applications and services, it is always important to develop necessary risk-control mechanisms and make contingency plans, which may need to be exercised before the mainnet deployment. The risk-control mechanisms should kick in at the very moment when the contracts are being deployed on mainnet. Please refer to Section 3 for details.

3 | Detailed Results

3.1 Improved Allowance Management in DeadDrop

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: DeadDrop
- Category: Coding Practices [7]
- CWE subcategory: CWE-561 [3]

Description

The Minterest protocol has a unique auto-liquidation mechanism that collects liquidation-related fee to the DeadDrop contract. This DeadDrop contract maintains a whitelisted set of swap routers that are used to swap liquidated assets. While examining the related swap routines, we notice the current allowance management may be improved.

To elaborate, we show below the related swapTokensForExactTokens() function. As the name indicates, this function is in essence a wrapper over the UniswapV2Router02::swapTokensForExactTokens() routine and can be used to swap for the intended amount of output token with the given amountInMax. With that, the current implementation approves the router for the spending of amountInMax. If the allowance is not used up, the router may still be entitled to transfer the remaining amount. With that, we suggest to improve the implementation by resetting the allowance to 0 after the swap operation.

```
    function swapTokensForExactTokens(
        uint256 amountInMax,
        uint256 amountOut,
        address[] memory path,
        IUniswapV2Router02 router,
        uint256 deadline
    ) external onlyRole(GUARDIAN) allowedRouter(router) {
        require(deadline >= block.timestamp, ErrorCodes.DD_EXPIRED_DEADLINE);
        IERC20 tokenIn = IERC20(path[0]);

        uint256 tokenInBalance = tokenIn.balanceOf(address(this));
        require(tokenInBalance >= amountInMax, ErrorCodes.INSUFFICIENT_LIQUIDITY);
```

```

        tokenIn.safeIncreaseAllowance(address(router), amountInMax);
        //slither-disable-next-line unused-return
        router.swapTokensForExactTokens(amountOut, amountInMax, path, address(this),
            deadline);

        uint256 newTokenInBalance = tokenIn.balanceOf(address(this));

        emit SwapTokensForExactTokens(
            amountInMax,
            tokenInBalance - newTokenInBalance,
            amountOut,
            router,
            path,
            deadline
        );
    }
```

Listing 3.1: `DeadDrop::swapTokensForExactTokens()`

Recommendation Revisit the above routine by resetting the allowance to 0 after the swap operation.

Status This issue has been fixed in the following commit: `46fdda0`.

3.2 Improved ERC20-Compliance Of MToken

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `MToken`
- Category: Coding Practices [7]
- CWE subcategory: CWE-1126 [1]

Description

Each asset supported by the `Minterest` protocol is integrated through a so-called `MToken` contract, which is an ERC20 compliant representation of balances supplied to the protocol. By minting `MToken`, users can earn interest through the `MToken`'s exchange rate, which increases in value relative to the underlying asset, and further gain the ability to use `MTokens` as collateral. There are currently two types of `MTokens`: `MToken` and `MEther`. In the following, we examine the ERC20 compliance of these `MTokens`.

The ERC20 specification defines a list of API functions (and relevant events) that each token contract is expected to implement (and emit). The failure to meet these requirements means the token contract cannot be considered to be ERC20-compliant. Naturally, as part of our audit, we

Table 3.1: Basic View-Only Functions Defined in The ERC20 Specification

Item	Description	Status
name()	Is declared as a public view function	✓
	Returns a string, for example "Tether USD"	✓
symbol()	Is declared as a public view function	✓
	Returns the symbol by which the token contract should be known, for example "USDT". It is usually 3 or 4 characters in length	✓
decimals()	Is declared as a public view function	✓
	Returns decimals, which refers to how divisible a token can be, from 0 (not at all divisible) to 18 (pretty much continuous) and even higher if required	✓
totalSupply()	Is declared as a public view function	✓
	Returns the number of total supplied tokens, including the total minted tokens (minus the total burned tokens) ever since the deployment	✓
balanceOf()	Is declared as a public view function	✓
	Anyone can query any address' balance, as all data on the blockchain is public	✓
allowance()	Is declared as a public view function	✓
	Returns the amount which the spender is still allowed to withdraw from the owner	✓

examine the list of API functions defined by the ERC20 specification and validate whether there exist any inconsistency or incompatibility in the implementation or the inherent business logic of the audited contract(s).

Our analysis shows that there is a minor ERC20 inconsistency or incompatibility issues found in the `MToken` contract. Specifically, when new `MToken` is minted, the current implementation emits the following event: `Transfer(address(this), lender, lendTokens)` (line 441). According to the `EIP-20` specification, "`A token contract which creates new tokens SHOULD trigger a Transfer event with the _from address set to 0x0 when tokens are created.`" With that, we suggest to follow the specification by emitting the event from the `address(0)`.

In the surrounding two tables, we outline the respective list of basic `view-only` functions (Table 3.1) and key `state-changing` functions (Table 3.2) according to the widely-adopted ERC20 specification. In addition, we perform a further examination on certain features that are permitted by the ERC20 specification or even further extended in follow-up refinements and enhancements (e.g., ERC777/ERC2222), but not required for implementation. These features are generally helpful, but may also impact or bring certain incompatibility with current DeFi protocols. Therefore, we consider it is important to highlight them as well. This list is shown in Table 3.3.

Recommendation Revise the `AToken` implementation to ensure its ERC20-compliance.

Table 3.2: Key `State-Changing` Functions Defined in The ERC20 Specification

Item	Description	Status
transfer()	Is declared as a public function	✓
	Returns a boolean value which accurately reflects the token transfer status	✓
	Reverts if the caller does not have enough tokens to spend	✓
	Allows zero amount transfers	✓
	Emits Transfer() event when tokens are transferred successfully (include 0 amount transfers)	✓
	Reverts while transferring to zero address	✓
transferFrom()	Is declared as a public function	✓
	Returns a boolean value which accurately reflects the token transfer status	✓
	Reverts if the spender does not have enough token allowances to spend	✓
	Updates the spender's token allowances when tokens are transferred successfully	✓
	Reverts if the from address does not have enough tokens to spend	✓
	Allows zero amount transfers	✓
	Emits Transfer() event when tokens are transferred successfully (include 0 amount transfers)	✓
	Reverts while transferring from zero address	✓
	Reverts while transferring to zero address	✓
approve()	Is declared as a public function	✓
	Returns a boolean value which accurately reflects the token approval status	✓
	Emits Approval() event when tokens are approved successfully	✓
	Reverts while approving to zero address	✓
Transfer() event	Is emitted when tokens are transferred, including zero value transfers	✓
	Is emitted with the from address set to $address(0x0)$ when new tokens are generated	✓
Approval() event	Is emitted on any successful call to approve()	✓

Table 3.3: Additional `Opt-in` Features Examined in Our Audit

Feature	Description	Opt-in
Deflationary	Part of the tokens are burned or transferred as fee while on transfer()/transferFrom() calls	—
Rebasing	The balanceOf() function returns a re-based balance instead of the actual stored amount of tokens owned by the specific address	—
Pausable	The token contract allows the owner or privileged users to pause the token transfers and other operations	✓
Blacklistable	The token contract allows the owner or privileged users to blacklist a specific address such that token transfers and other operations related to that address are prohibited	—
Mintable	The token contract allows the owner or privileged users to mint tokens to a specific address	✓
Burnable	The token contract allows the owner or privileged users to burn tokens of a specific address	✓

Status This issue has been fixed in the following commit: `dd22fbb`.

3.3 Possible Front-Running For Unintended Payment In repayBorrowBehalf()

- ID: PVE-003
- Severity: Low
- Likelihood: Medium
- Impact: Low
- Target: `MToken`
- Category: Time and State [9]
- CWE subcategory: CWE-663 [4]

Description

As mentioned earlier, the `Minterest` protocol is in essence an over-collateralized lending pool that has the lending functionality and supports a number of normal lending functionalities for supplying and borrowing users, i.e., `mint()/redeem()` and `borrow()/repay()`. In the following, we examine one specific functionality, i.e., `repay()`.

To elaborate, we show below the core routine `repayBorrowFresh()` that actually implements the main logic behind the `repay()` routine. This routine allows for repaying partial or full current borrowing balance. It is interesting to note that the protocol supports the payment on behalf of another borrowing user (via `repayBorrowBehalf()`). And the `repayBorrowFresh()` routine supports the corner case when the given amount is larger than the current borrowing balance. In this corner case, the protocol assumes the intention for a full repayment.

```
    function repayBorrowFresh(
        address payer,
        address borrower,
        uint256 repayAmount,
        bool isERC20based
    ) internal nonReentrant returns (uint256 actualRepayAmount) {
        /* Fail if repayBorrow not allowed */
        supervisor.beforeRepayBorrow(address(this), borrower);

        /* Verify market's block number equals current block number */
        require(accrualBlockNumber == getBlockNumber(), ErrorCodes.MARKET_NOT_FRESH);

        /* We fetch the amount the borrower owes, with accumulated interest */
        uint256 borrowBalance = borrowBalanceStoredInternal(borrower);

        if (repayAmount == type(uint256).max) {
            repayAmount = borrowBalance;
        }

        /////////////////////////
        // EFFECTS & INTERACTIONS

        /*
         * We call doTransferIn for the payer and the repayAmount
         *  Note: The mToken must handle variations between ERC-20 and ETH underlying.
         *  On success, the mToken holds an additional repayAmount of cash.
         *  doTransferIn reverts if anything goes wrong, since we can't be sure if side
              effects occurred.
         *   it returns the amount actually transferred, in case of a fee.
         */
        // slither-disable-next-line reentrancy-eth
        if (isERC20based) {
            actualRepayAmount = doTransferIn(payer, repayAmount);
        } else {
            actualRepayAmount = repayAmount;
        }

        /*
         * We calculate the new borrower and total borrow balances, failing on underflow
              :
         *  accountBorrowsNew = accountBorrows - actualRepayAmount
         *  totalBorrowsNew = totalBorrows - actualRepayAmount
         */
        uint256 accountBorrowsNew = borrowBalance - actualRepayAmount;
        uint256 totalBorrowsNew = totalBorrows - actualRepayAmount;

        accountBorrows[borrower].principal = accountBorrowsNew;
        accountBorrows[borrower].interestIndex = borrowIndex;
        totalBorrows = totalBorrowsNew;

        emit RepayBorrow(payer, borrower, actualRepayAmount, accountBorrowsNew,
            totalBorrowsNew);
```

```
    }
```

Listing 3.2: `MToken::repayBorrowFresh()`

This is a reasonable assumption, but our analysis shows this assumption may be taken advantage of to launch a front-running `borrow()` operation, resulting in a higher borrowing balance for repayment. To avoid this situation, it is suggested to disallow the repayment amount of -1 to imply the full repayment. In fact, it is always suggested to use the exact payment amount in the `repayBorrowBehalf()` case.

Recommendation Revisit the generous assumption of using repayment amount of -1 as the indication of full repayment.

Status This issue has been confirmed. Considering the given amount is the choice from the repayer, the team decides to leave it as is.

3.4 Possible Sandwich/MEV Attacks For Drip Collection

- ID: PVE-004
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `BuybackDripper`, `Buyback`
- Category: Time and State [9]
- CWE subcategory: CWE-663 [4]

Description

As part of the incentive mechanisms, the `Minterest` protocol has the `BuybackDripper` that distributes a token to buyback at a defined drip rate. The participating user is entitled to receive pro-rata drip distribution based on the weight of the associated user account. Our analysis shows this mechanism may be abused to steal most drip distribution.

To elaborate, we show below the `drip()` routine. As the name indicates, the function drips tokens to buyback with the defined drip rate. By design, it cannot be called more than once per hour. We notice this function is permissionless and open to public. Internally, it invokes the `buyback()` routine (line 94), which basically re-computes and saves the new `shareAccMantissa` state to record the accumulated reward index.

```
71      function drip() external {
72          uint256 timeUnits = getTime();
73          uint256 timeSinceDrip = timeUnits - previousDripTime;
74          require(timeSinceDrip > 0, ErrorCodes.TOO_EARLY_TO_DRIP);

76          // Reset period if last drip was older than period duration
77          if (timeSinceDrip >= periodDuration) {
```

```
            previousDripTime = timeUnits;
            resetPeriod(timeUnits);
            return;
        }

        uint256 nextPeriodStart = periodStart + periodDuration;

        uint256 dripUntil = Math.min(timeUnits, nextPeriodStart);
        uint256 dripDuration = dripUntil - previousDripTime;
        uint256 toDrip = dripDuration * dripPerHour;
        previousDripTime = dripUntil;

        if (dripUntil >= nextPeriodStart) {
            resetPeriod(nextPeriodStart);
        }

        buyback.buyback(toDrip);
    }
```

Listing 3.3: `BuybackDripper::drip()`

```
    function buyback(uint256 amount_) external onlyRole(DISTRIBUTOR) {
        require(amount_ > 0, ErrorCodes.NOTHING_TO_DISTRIBUTE);
        require(weightSum > 0, ErrorCodes.NOT_ENOUGH_PARTICIPATING_ACCOUNTS);

        uint256 shareMantissa = (amount_ * SHARE_SCALE) / weightSum;
        shareAccMantissa = shareAccMantissa + shareMantissa;

        emit NewBuyback(amount_, shareMantissa);

        mnt.safeTransferFrom(msg.sender, address(this), amount_);
    }
```

Listing 3.4: `Buyback::buyback()`

With that, it is possible to have a sandwich scenario where a malicious actor may flash to borrow a large amount of asset to stake to increase the weight of a controlled account, then invoke the above open `drip()`, and next unstake and repay the flashloan. By doing so, the malicious actor may simply have the most share of the tokens that are just distributed via `drip()`.

Recommendation Improve the above drip mechanism to ensure the user staked funds are locked for a certain period to thwart possible flashloan-assisted MEV attacks.

Status This issue has been fixed in the following commit: `b26d717`.

3.5 Trust Issue of Admin Keys

- ID: PVE-005
- Severity: Medium
- Likelihood: Low
- Impact: High
- Target: `Multiple Contracts`
- Category: Security Features [6]
- CWE subcategory: CWE-287 [2]

Description

In the `Minterest` protocol, there is a privileged account (with the role of `DEFAULT_ADMIN_ROLE`) that plays a critical role in governing and regulating the system-wide operations (e.g., parameter setting and marketing adjustment). It also has the privilege to control or govern the flow of assets managed by this protocol. Our analysis shows that the privileged account needs to be scrutinized. In the following, we examine the privileged account and their related privileged accesses in current contracts.

```
    function setBuyback(Buyback newBuyback) external onlyRole(DEFAULT_ADMIN_ROLE) {
        Buyback oldBuyback = buyback;
        buyback = newBuyback;
        emit NewBuyback(oldBuyback, newBuyback);
    }

    /**
     * @notice Sets a new emissionBooster for the supervisor
     * @dev Admin function to set a new EmissionBooster. Can only be installed once.
     */
    function setEmissionBooster(EmissionBooster _emissionBooster) external onlyRole(
        DEFAULT_ADMIN_ROLE) {
        require(Address.isContract(address(_emissionBooster)), ErrorCodes.
            CONTRACT_DOES_NOT_SUPPORT_INTERFACE);
        require(address(emissionBooster) == address(0), ErrorCodes.CONTRACT_ALREADY_SET)
            ;
        emissionBooster = _emissionBooster;
        emit NewEmissionBooster(emissionBooster);
    }

    /// @notice function to set BDSystem contract
    /// @param newBDSystem_ new Business Development system contract address
    function setBDSystem(BDSystem newBDSystem_) external onlyRole(DEFAULT_ADMIN_ROLE) {
        BDSystem oldBDSystem = bdSystem;
        bdSystem = newBDSystem_;
        emit NewBusinessDevelopmentSystem(oldBDSystem, newBDSystem_);
    }

    /*
     * @notice Sets a new whitelist for the supervisor
     * @dev Admin function to set a new whitelist
     */
```

```
    function setWhitelist(WhitelistInterface newWhitelist_) external onlyRole(
        DEFAULT_ADMIN_ROLE) {
        WhitelistInterface oldWhitelist = whitelist;
        whitelist = newWhitelist_;
        emit NewWhitelist(oldWhitelist, newWhitelist_);
    }
```

Listing 3.5: Example `Setters` in the `Supervisor`

We understand the need of the privileged functions for proper contract operations, but at the same time the extra power to these privileged accounts may also be a counter-party risk to the contract users. Therefore, we list this concern as an issue here from the audit perspective and highly recommend making these privileges explicit or raising necessary awareness among protocol users.

Recommendation Make the privileges explicit to the protocol users.

Status This issue has been mitigated. The team confirms that the `DEFAULT_ADMIN_ROLE` is meant to be here only during the setup of the protocol and initial stage of its growth. The first big `DAO` proposal is planned to be a substitute of all the `DEFAULT_ADMIN_ROLE` addresses with the `DAO` contract address as soon as it will be possible to make the protocol self-governed by the community.

3.6 Accommodation of Non-ERC20-Compliant Tokens

- ID: PVE-006
- Severity: Low
- Likelihood: Low
- Impact: High
- Target: `Treasury`, `MNTSource`
- Category: Business Logic [8]
- CWE subcategory: CWE-841 [5]

Description

Though there is a standardized ERC-20 specification, many token contracts may not strictly follow the specification or have additional functionalities beyond the specification. In the following, we examine the `transfer()` routine and related idiosyncrasies from current widely-used token contracts.

In particular, we use the popular token, i.e., `ZRX`, as our example. We show the related code snippet below. On its entry of `transfer()`, there is a check, i.e., `if (balances[msg.sender] >= _value && balances[_to] + _value >= balances[_to])`. If the check fails, it returns `false`. However, the transaction still proceeds successfully without being reverted. This is not compliant with the ERC20 standard and may cause issues if not handled properly. Specifically, the ERC20 standard specifies the following: *"Transfers _value amount of tokens to address _to, and MUST fire the Transfer event. The function SHOULD throw if the message caller's account balance does not have enough tokens to spend."*

```
    function transfer(address _to, uint _value) returns (bool) {
        //Default assumes totalSupply can't be over max (2^256 - 1).
        if (balances[msg.sender] >= _value && balances[_to] + _value >= balances[_to]) {
            balances[msg.sender] -= _value;
            balances[_to] += _value;
            Transfer(msg.sender, _to, _value);
            return true;
        } else { return false; }
    }

    function transferFrom(address _from, address _to, uint _value) returns (bool) {
        if (balances[_from] >= _value && allowed[_from][msg.sender] >= _value &&
            balances[_to] + _value >= balances[_to]) {
            balances[_to] += _value;
            balances[_from] -= _value;
            allowed[_from][msg.sender] -= _value;
            Transfer(_from, _to, _value);
            return true;
        } else { return false; }
    }
```

Listing 3.6: ZRX.sol

Because of that, a normal call to `transfer()` is suggested to use the safe version, i.e., `safeTransfer()`, In essence, it is a wrapper around ERC20 operations that may either throw on failure or return false without reverts. Moreover, the safe version also supports tokens that return no value (and instead revert or throw on failure). Note that non-reverting calls are assumed to be successful. Similarly, there is a safe version of `approve()/transferFrom()` as well, i.e., `safeApprove()/safeTransferFrom()`.

In the following, we show the `sendToAddress()` routine in the `Treasury` contract. If the `USDT` token is supported as `currency`, the unsafe version of `require(currency.transfer(to, amount))` (line 57) may revert as there is no return value in the `USDT` token contract's `transfer()/transferFrom()` implementation (but the `IERC20` interface expects a return value)!

```
47      function sendToAddress(
48          address to,
49          uint256 amount,
50          IERC20 currency
51      ) external onlyOwner onlyApprovedReceiver(to) {
52          require(amount > 0, ErrorCodes.INCORRECT_AMOUNT);
53          require(address(currency) != address(0), ErrorCodes.
                CURRENCY_ADDRESS_CANNOT_BE_ZERO);
54          require(currency.balanceOf(address(this)) >= amount, ErrorCodes.
                INSUFFICIENT_FUNDS);
55
56          //slither-disable-next-line reentrancy-events
57          require(currency.transfer(to, amount));
58          emit TokenSent(to, amount, currency);
59      }
```

Listing 3.7: `FantomMint::sendToAddress()`

Recommendation Accommodate the above-mentioned idiosyncrasy about ERC20-related `approve()/transfer()/transferFrom()`. Note the `MNTSource::drip()` routine can be similarly improved.

Status This issue has been fixed in the following commit: `6a817bb`.

3.7 Improved Gas Efficiency in EmissionBooster::enableTiersInternal()

- ID: PVE-007
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Multiple Contracts`
- Category: Coding Practices [7]
- CWE subcategory: CWE-561 [3]

Description

Since version 0.6.5, `Solidity` introduces the feature of declaring a state as `immutable`. An `immutable` state variable can only be assigned during contract creation, but will remain constant throughout the life-time of a deployed contract. The main benefit of declaring a state as `immutable` is that reading the state is significantly cheaper than reading from regular storage, since it is not stored in storage anymore. Instead, an `immutable` state will be directly inserted into the runtime code.

This feature is introduced based on the observation that the reading and writing of storage-based contract states are gas-expensive. Therefore, it is always preferred if we can reduce, if not eliminate, storage reading and writing as much as possible. Those state variables that are written only once are candidates of `immutable` states under the condition that each fits the pattern, i.e., "a constant, once assigned in the constructor, is read-only during the subsequent operation."

While examining all the state variables defined in the `MNTSource` contract, we observe there are several variables (e.g., `dripStart`, `dripRate`, `token`, and `target`) that need not to be updated dynamically. They can be declared as `immutable` for gas efficiency.

```
contract MNTSource {
    /// @notice The block number when the MNTSource started (immutable)
    uint256 public dripStart;

    /// @notice Tokens per block that to drip to target (immutable)
    uint256 public dripRate;

    /// @notice Reference to token to drip (immutable)
    IERC20 public token;

    /// @notice Target to receive dripped tokens (immutable)
    address public target;
```

```

    /// @notice Amount that has already been dripped
    uint256 public dripped;

    ...
}
```

Listing 3.8: `MNTSource`

In addition, the protocol has a core `EmissionBooster` contract that is designed to apply the unique boost mechanism. This contract has an internal function `enableTiersInternal()` and its current implementation may repeatedly write the same storage state with the same value (lines 263-264). For gas efficiency, the repeated writes on the same storage need to be avoided.

```
238     function enableTiersInternal(uint256[] memory tiersForEnabling) internal {
239         uint32 currentBlock = uint32(_getBlockNumber());
240 
241         // For each tier of tiersForEnabling set startBlock
242         for (uint256 i = 0; i < tiersForEnabling.length; i++) {
243             uint256 tier = tiersForEnabling[i];
244             require(tier != 0, ErrorCodes.EB_ZERO_TIER_CANNOT_BE_ENABLED);
245             require(tierExists(tier), ErrorCodes.EB_TIER_DOES_NOT_EXIST);
246             require(!isTierActive(tier), ErrorCodes.EB_ALREADY_ACTIVATED_TIER);
247             require(currentBlock < tiers[tier].endBlock, ErrorCodes.
                    EB_END_BLOCK_MUST_BE_LARGER_THAN_CURRENT);
248             tiers[tier].startBlock = currentBlock;
249         }
250 
251         _rebuildCheckpoints();
252 
253         // For all markets update mntSupplyIndex and mntBorrowIndex, and set
                marketSpecificData index
254         MToken[] memory markets = supervisor.getAllMarkets();
255         for (uint256 i = 0; i < markets.length; i++) {
256             MToken market = markets[i];
257             tierToBeUpdatedSupplyIndex[market] = getNextTierToBeUpdatedIndex(market,
                    true);
258             tierToBeUpdatedBorrowIndex[market] = getNextTierToBeUpdatedIndex(market,
                    false);
259             // slither-disable-next-line reentrancy-events,calls-loop
260             (uint224 mntSupplyIndex, uint224 mntBorrowIndex) = supervisor.
                    updateAndGetMntIndexes(market);
261             for (uint256 index = 0; index < tiersForEnabling.length; index++) {
262                 uint256 tier = tiersForEnabling[index];
263                 marketSupplyIndexes[market][currentBlock] = mntSupplyIndex;
264                 marketBorrowIndexes[market][currentBlock] = mntBorrowIndex;
265                 emit TierEnabled(market, tier, currentBlock, mntSupplyIndex,
                        mntBorrowIndex);
266             }
267         }
```

```
    }
```

Listing 3.9: EmissionBooster::enableTiersInternal()

Recommendation Improve the gas efficiency in the above routines.

Status This issue has been fixed in the following commit: `fb74afe`.

4 Conclusion

In this audit, we have analyzed the design and implementation of the `Minterest` protocol, which is a decentralised lending protocol. It comes with a unique economic model where the protocol itself captures 100% of the value created from its functions, including interest, flash loan, and auto-liquidation fees. This value is exchanged via an automated `Buyback` process for its native `MNT` token which is then distributed to its users in return for their participation in the protocol's governance. The current code base is well structured and neatly organized. Those identified issues are promptly confirmed and addressed.

References

[1] MITRE. CWE-1126: Declaration of Variable with Unnecessarily Wide Scope. https://cwe.mitre.org/data/definitions/1126.html.

[2] MITRE. CWE-287: Improper Authentication. https://cwe.mitre.org/data/definitions/287.html.

[3] MITRE. CWE-561: Dead Code. https://cwe.mitre.org/data/definitions/561.html.

[4] MITRE. CWE-663: Use of a Non-reentrant Function in a Concurrent Context. https://cwe.mitre.org/data/definitions/663.html.

[5] MITRE. CWE-841: Improper Enforcement of Behavioral Workflow. https://cwe.mitre.org/data/definitions/841.html.

[6] MITRE. CWE CATEGORY: 7PK - Security Features. https://cwe.mitre.org/data/definitions/254.html.

[7] MITRE. CWE CATEGORY: Bad Coding Practices. https://cwe.mitre.org/data/definitions/1006.html.

[8] MITRE. CWE CATEGORY: Business Logic Errors. https://cwe.mitre.org/data/definitions/840.html.

[9] MITRE. CWE CATEGORY: Concurrency. https://cwe.mitre.org/data/definitions/557.html.

[10] MITRE. CWE VIEW: Development Concepts. https://cwe.mitre.org/data/definitions/699.html.

[11] OWASP. Risk Rating Methodology. https://www.owasp.org/index.php/OWASP_Risk_Rating_Methodology.

[12] PeckShield. PeckShield Inc. https://www.peckshield.com.