PRIVATE

Code Assessment

of the M Extensions Smart Contracts

August 26, 2025

Produced for

M^ZERO LABS_

by

CHAINSECURITY

Contents

1 Executive Summary

Dear all,

Thank you for trusting us to help M^ZERO Labs with this security audit. Our executive summary provides an overview of subjects covered in our audit of the latest reviewed contracts of M Extensions according to Scope to support you in forming an opinion on their security risks.

M^ZERO Labs implements a suite of different ERC-20 stablecoin contracts designed to wrap the existing `$M` token into non-rebasing tokens, for better composability in the broader DeFi ecosystem. These contracts, called *extensions*, differ in how they treat and redistribute the yield generated by their `$M` balance.

Additionally, M^ZERO Labs offers a `SwapFacility` contract, that will act as a gateway towards these extensions, being the only privileged address allowed to wrap and unwrap `$M` tokens.

The most critical subjects covered in our audit are asset solvency, functional correctness, and precision of arithmetic operations. Security regarding all the aforementioned subjects is high, after all outstanding issues have been addressed.

The general subjects covered are documentation, gas efficiency, and the integration of the wrapper into the existing system. Security regarding all the aforementioned subjects is generally good.

In summary, we find that the codebase provides a high level of security.

It is important to note that security audits are time-boxed and cannot uncover all vulnerabilities. They complement but don't replace other vital measures to secure a project.

The following sections will give an overview of the system, our methodology, the issues uncovered, and how they have been addressed. We are happy to receive questions and feedback to improve our service.

Sincerely yours,

ChainSecurity

1.1 Overview of the Findings

Below we provide a brief numerical overview of the findings and how they have been addressed.

Critical-Severity Findings	0
High-Severity Findings	1
• Code Corrected	1
Medium-Severity Findings	5
• Code Corrected	4
• Acknowledged	1
Low-Severity Findings	5
• Code Corrected	2
• Acknowledged	3

2 Assessment Overview

In this section, we briefly describe the overall structure and scope of the engagement, including the code commit which is referenced throughout this report.

2.1 Scope

The assessment was performed on the source code files inside the M Extensions repository based on the documentation files. The table below indicates the code versions relevant to this report and when they were received.

V	Date	Commit Hash	Note
1	23 Jun 2025	66eb4710e73ce1ffcf198b593e9e81848f9385cc	Initial Version
2	17 Jul 2025	ba39e694aa7bfffd5138a0ead9f9cb7438c7929a	Version with fixes
3	31 Jul 2025	0bf9cadd2002a0fa656ba4e34c2cb90eb73a2da9	Second round of fixes
4	13 Aug 2025	011f84f0f6a701a9796fcac1ad29896c60b65344	Freezable and SwapFacility permission logic update
5	26 Aug 2025	909c536deac54de5a5bc3305f57e310c327fb441	Balance check and NatSpec updates

For the solidity smart contracts, the compiler version `0.8.26` was chosen.

The following contracts are in the scope of the review:

```
MExtension.sol
components:
    Blacklistable.sol
libs:
    IndexingMath.sol
projects:
    earnerManager:
        MEarnerManager.sol
    yieldToAllWithFee:
        MSpokeYieldFee.sol
        MYieldFee.sol
    yieldToOne:
        MYieldToOne.sol
swap:
    SwapFacility.sol
    UniswapV3SwapAdapter.sol
```

After V2, the scope has been updated as follows:

Added:

```
swap:
    ReentrancyLock.sol
```

After V4, the scope has been updated as follows:

Removed:

```
src/components/Blacklistable.sol
```

Added:

```
src/components/Freezable.sol
```

2.1.1 Excluded from scope

Any contracts not explicitly listed above are out of the scope of this review. Third-party libraries are out of the scope of this review. More specifically, `openzeppelin-contracts-upgradeable`, `openzeppelin-contracts` and `uniswap-v4-periphery` are expected to work as intended and are out of the scope of this review. The Uniswap V3 protocol and the `SwapRouter02` router are expected to work as intended and are out of the scope of this review. The code related to the core protocol (`MToken`) and `common/` is out of the scope of this review but a dedicated review can be found at https://www.chainsecurity.com/security-audit/m-zerprotocol-and-governance.

2.2 System Overview

This system overview describes the initially received version (Version 1) of the contracts as defined in the Assessment Overview.

Furthermore, in the findings section, we have added a version icon to each of the findings to increase the readability of the report.

M^ZERO Labs offers a suite of different ERC-20 stablecoin contracts designed to wrap the existing `$M` token into non-rebasing tokens, for better composability in the broader DeFi ecosystem. These contracts, called *extensions*, differ in how they treat and redistribute the yield generated by their `$M` balance.

Additionally, M^ZERO Labs will deploy a `SwapFacility` contract, that will act as a gateway towards these extensions, being the only privileged address allowed to wrap and unwrap `$M` tokens.

2.2.1 Extensions

Three different extensions are in scope for this review: `MYieldToOne`, `MEarnerManager`, and the pair `MYieldFee` - `MSpokeYieldFee`.
They will *not* have minting privileges for the `$M` token, therefore the wrap/unwrap functionality is implemented with a lock-unlock mechanism. They will, however, be whitelisted as `$M` earners (more on that later), therefore their locked `$M` balance will generate yield: this yield gets represented by additional wrapped tokens to be minted by an ad-hoc function, called `claimYield()` or some variant thereof.
It is the intention of M^ZERO Labs that these extensions be deployed by selected partner entities and individually whitelisted in the `SwapFacility`.
All of the extensions are meant to be deployed behind transparent upgradable proxies.

2.2.1.1 MExtension

This is an abstract contract that serves as a basis for the three concrete extensions.
As a matter of public-facing functions, it defines little more than the access control, token flow, and exchange logic for the `wrap()` and `unwrap()` functions: only the `SwapFacility` can call them; `$M` tokens only flow to/from the `SwapFacility`; `$M` tokens are always exchanged one-to-one for the

wrapped version. These functions are inherited without override: only the associated `_mint()` and `_burn()` logic is left to be implemented by the concrete inheriting contracts. The implementation of the internal accounting (`balanceOf()`, `_update()`) is delegated to the concrete implementation as well.
Furthermore, it declares optional hooks, to be defined by extensions needing to intercept particular user actions; these are: `_beforeWrap()`, `_beforeUnwrap()`, `_beforeApprove()`, `_beforeTransfer()`.

2.2.1.2 MYieldToOne

This is the simplest of the three wrappers. All the yield goes to a single, configurable `yieldRecipient` address. It also features a blacklist, restricting token movements among ordinary users.
A privileged `yieldRecipientManager` address can set the `yieldRecipient` at will; another privileged `blacklistManager` address can add and remove arbitrary addresses to/from the blacklist. When a new `yieldRecipient` recipient is set, the yield is not automatically claimed for the old recipient.
The permissionless `claimYield()` function computes the "pending" yield as the difference between the contract's own `$M` balance and the `totalSupply()` of wrapped tokens: this amount of *wrapped* tokens is then minted to the `yieldRecipient`.
All four hooks are implemented to enforce the blacklist on every user action (`wrap()`, `unwrap()`, `approve()`, `transfer()`, `transferFrom()`, `permit()`, `transferWithAuthorization()`).
This extension can seamlessly adapt to the case where the `$M` governance revokes its earning status; no particular action is required to update its internal accounting.

2.2.1.3 MEarnerManager

This wrapper redistributes yield to all of its users, minus a fee determined by a per-address `feeRate` taken as a percentage of the accrued yield. It features a whitelist of addresses allowed to hold and move tokens.
A privileged `earnerManager` address can set the global `feeRecipient` address, as well as the individual fee rates; it also manages the whitelist. If an address is un-whitelisted, its funds get frozen, and its fee rate is set to 100% (this allows its future yield not to be frozen, and to go entirely to the `feeRecipient`). By default, all accounts have a `feeRate` of 0% unless whitelisted with a different fee rate, or un-whitelisted.
Anyone can permissionlessly claim yield on behalf of any other account, using the `claimFor(address account)` function. The pending yield is computed by using the `currentIndex()` of the `$M` token: the result is equal to the `$M` yield earned by the extension contract, "induced" by the `account`'s deposit, since the last claim (this effectively means that, when all users claim, all yield is redistributed). The corresponding quantity of *wrapped* tokens is minted to the `account`, and then the appropriate cut on the yield (determined by the `account`'s `feeRate`) is explicitly transferred to the `feeRecipient`.
No function exists to "sweep" all the "pending fee" in one go. Since users all have different `feeRates`, the only way to collect all the pending fees is to iteratively call `claimFor()` over all users. For this to be manageable, the whitelist is expected to stay within a reasonable size.
All four hooks are implemented to enforce the whitelist on every user action (`wrap()`, `unwrap()`, `approve()`, `transfer()`, `transferFrom()`, `permit()`, `transferWithAuthorization()`). In particular, the token source (`msgSender()` of the `SwapFacility`) when wrapping must also be whitelisted, arbitrary addresses cannot wrap for a whitelisted address.
This extension *cannot* adapt to the case where the `$M` governance revokes its earning status. It will keep crediting undue/unredeemable yield indefinitely to its users (always using the growing `currentIndex()` of the `$M` token), eventually leading to insolvency, and a bank run. This situation is expected to be remedied with an upgrade.

2.2.1.4 MYieldFee

This extension credits the same yield rate to all its users; this is the `$M` token's `earnerRate()`, discounted by a configurable `feeRate`: the yield not credited to users makes up the fee.

A privileged `feeManager` address can set the `feeRate` and the `feeRecipient`. Another privileged `claimRecipientManager` address can set a `claimRecipient` for each user: this effectively redirects the user's yield to an admin-chosen address.
A permissionless `updateIndex()` function, analogous to the one in the `$M` token, exists to "pivot around" the current index, and apply a new earner rate from this point on. This function needs to be called promptly, whenever the `earnerRate()` changes on `$M`.
Anyone can permissionlessly claim yield on behalf of any other account, using the `claimYieldFor(address account)` function. The pending yield is calculated by using the `currentIndex()` recomputed internally using the discounted rate. This yield is then credited to the `account`, and possibly forwarded to its `claimRecipient`, if one is set.
The permissionless function `claimFee()` computes all the pending fee as the difference between the `$M` balance of the extension contract, and the *projected total supply* (i.e. a preview of the total supply, should every user claim its yield). The fee is simply minted to the fee recipient.
None of the hooks are implemented, since no blacklist/whitelist checks need to be enforced.
This extension can handle the case where the `$M` governance revokes its earning status. The permissionless function `disableEarning()` needs to be called promptly, as soon as the extension contract stops being an `$M` earner; this function sets the rate to 0, effectively stopping undue yield from being credited to users. Should the extension contract regain the `$M` earner status, anyone can call the permissionless function `enableEarning()` in order to again start using `$M`'s `earnerRate()` (discounted by the fee).

2.2.1.5 MSpokeYieldFee

This is a specialized version of `MYieldFee` that can be deployed on supported EVM sidechains (e.g. Arbitrum, Optimism) where the `$M` token is deployed. These chains are called *spoke* chains.
While `MYieldToOne` and `MEarnerManager` can be deployed to spoke chains without modifications, `MYieldFee` needs to query the current `earnerRate()` on the `$M` token, which is absent in spoke chains deployments. On spoke chains, the `$M` index does not continuously grow according to a rate; instead, the `currentIndex()` is periodically bridged over from L1 and applied immediately. This results in discrete jumps at every bridging operation.
The `MSpokeYieldFee` inherits from `MYieldFee`, overriding some of its internal functions so as to mimic the step-wise behavior of the bridged `$M` index. First, the `earnerRate()` is queried from an ad-hoc `rateOracle`, set at initialization time; notice that the implementation of the rate oracle is out of scope for this review. Second, the exponential computed in `currentIndex()` does not utilize `block.timestamp`, but the timestamp of the last `$M` index bridging. This ensures that `currentIndex()` follows the same jumps as in the `$M` token, minus the fee.
The rest of the functionality is identical to the `MYieldFee`.

2.2.2 SwapFacility

The `SwapFacility` is deployed behind a transparent proxy and is the only entity allowed to call `wrap()/unwrap()` on the extensions, it acts as the hub for:

- swapping between the extensions with `swap()`. It simply unwraps from the source and wraps into the destination extension.
- wrapping `MToken` into one of the extensions with `swapInM()` and `swapInMWithPermit()`. The `MToken` will be transferred to the `SwapFacility` and then wrapped into the desired extension.
- swapping some whitelisted token to one of the extension tokens, or the other way around, via `Uniswap V3` (through the `UniswapV3SwapAdapter`): functions `swapInToken()` and `swapOutToken()`.
- unwrapping `MToken` from one of the extensions with `swapOutM()`. Only whitelisted addresses are allowed to do it, as regular users should not directly hold the `MToken` in practice, but rather interact with one of the wrapped versions.

All the wrapping/unwrapping actions described above are subject to the limitations imposed by the source and/or destination extensions, e.g. blacklist, whitelist, for the addresses involved.

2.2.3 UniswapV3SwapAdapter

This contract is immutable and serves as a helper to interact with Uniswap V3's `SwapRouter02` to swap from/to the base token (wrapped `MToken`) to/from some whitelisted token. The token whitelist is managed by an admin address having the `DEFAULT_ADMIN_ROLE`. The contract exposes two main functions: `swapOut()` and `swapIn()`. The two functions can be given a `path`, following the token-fee-token structure from UniswapV3, for multi-hop swaps. An empty path is possible and will default to a single swap on the token pair `baseToken/[in-out]put` with fee `0.01%`.

2.2.4 Changes in V2

- `MEarnerManager` has been fixed in its internal accounting, so that now it can gracefully handle the event of being removed from the earners list, without triggering a bank run or requiring an upgrade.
- The `MExtension` burns the shares from the `SwapFacility` instead of the `SwapFacility.msgSender()`.
- In the `MYieldToOne` the yield is automatically claimed for the old yield recipient before the new recipient is set.
- The `Lock` implemented by `SwapFacility` allowing the contract to reenter itself has been replaced by the new `ReentrancyLock`.
- The `UniswapV3SwapAdapter` contract now implements the `ReentrancyLock` from `uniswap-v4-periphery`.
- The `SwapFacility` no longer offers swap functions from/to arbitrary tokens, and therefore no longer depends on the `UniswapV3SwapAdapter` (aside from the `ReentrancyLock` component).
- The `UniswapV3SwapAdapter` is now a standalone helper contract that exposes user-facing swap functions from/to arbitrary tokens.

2.2.5 Changes in V4

- `Blacklistable` has been renamed to `Freezable`, but the functionality stays the same.
- `MYieldToOne` implements a new hook `_beforeClaimYield()` called at the beginning of `claimYield()`. The hook has no functionality in `MYieldToOne` but can be overridden by contracts extending it.
- The `SwapFacility` has a more fine-grained access control. The extensions can be marked as *permissioned* by the `DEFAULT_ADMIN_ROLE`. A permissioned extension can only be swapped from and to by a swapper address that was explicitly whitelisted for that extension by the `DEFAULT_ADMIN_ROLE`, overriding the `M_SWAPER_ROLE` role. The updated behavior of the swapping functions are listed below:
    - `swap()/swapWithPermit()`: both extensions must be approved as before, both extensions must be *not permissioned*
    - `swapInM()/swapInMWithPermit()`: the `extensionOut` must be approved as before, the caller must be explicitly whitelisted if the `extensionOut` is *permissioned*, otherwise the caller must have the `M_SWAPPER_ROLE`. Previously, any `MToken` holder could use this function to swap to an authorized extension.
    - `swapOutM()/swapOutMWithPermit()`: the `extensionIn` must be approved as before, the caller must be explicitly whitelisted if the `extensionIn` is *permissioned* (new), otherwise the caller must have the `M_SWAPPER_ROLE` (as before).

2.3 Roles and Trust Model

- Users: not trusted.
- Proxy admins: fully trusted. They are trusted to manage the proxies implementation in the best interest of the users and in a non-adversarial manner. If malicious or compromised, they can upgrade the implementations of the wrappers and/or `SwapFacility` and steal all the locked `MTokens`.
- Bearers of `DEFAULT_ADMIN_ROLE` in general: fully trusted. They are trusted to manage the internal roles and permissions of the contracts in the best interest of the users and in a non-adversarial manner. If malicious or compromised, they can, for example, distribute critical roles to lock (DOS) users' funds and collect their yield (`MEarnerManager` and `MYieldToOne`).
- Bearers of the `DEFAULT_ADMIN_ROLE` in `UniswapV3SwapAdapter`: semi-trusted. They are trusted to manage the whitelist of tokens to facilitate swapping from and to the `M` ecosystem in the best interest of the users and in a non-adversarial manner. If malicious or compromised, DOS the the `swapInToken()` and `swapOutToken()` of the `SwapFacility`.
- Bearers of the `M_SWAPPER_ROLE` in `SwapFacility`: semi-trusted. They are trusted to set a correct whitelist of addresses who are allowed to call `swapOutM()`. If set too lax, the `$M` token may start circulating among regular users. If set too strictly, legitimate `$M` holders might be hampered in their operations.
- Bearers of the `EARNER_MANAGER_ROLE` in `MEarnerManager`: fully-trusted. They are trusted to manage the whitelist and the applied fees in the best interest of the users and in a non-adversarial manner. If malicious or compromised, they can lock (DOS) user's funds and redirect the yield to an address they control.
- Bearers of the `FEE_MANAGER_ROLE` in `MYieldFee`: semi-trusted. They are trusted to manage the applied fee in the best interest of the users and in a non-adversarial manner. If malicious or compromised, they can steal the yield by setting the fee to 100% and redirect the yield to an address they control but cannot DOS users.
- Bearers of the `CLAIM_RECIPIENT_MANAGER_ROLE` in `MYieldFee`: semi-trusted. They are trusted to manage the users' claim recipients in the best interest of the users and in a non-adversarial manner. If malicious or compromised, they can redirect the yield to an address they control but cannot DOS users.
- Bearers of the `FREEZE_MANAGER_ROLE` in `MYieldToOne`: fully trusted. They are trusted to manage the freeze-list in the best interest of the users and the entity collecting the yield, and in a non-adversarial manner. If malicious or compromised, they can lock (DOS) user's funds or freeze the `feeRecipient`.
- Bearers of the `YIELD_RECIPIENT_MANAGER_ROLE` in `MYieldToOne`: semi-trusted. They are trusted to manage the yield recipient address in the best interest of the entity collecting the yield. If malicious or compromised, they can redirect the yield to an address they control but cannot DOS users.

Other assumptions:

- `SwapFacility` will never be added to the earner list, nor be upgraded to expose a function calling `MToken.startEarning()`. The same goes for `UniswapV3SwapAdapter`.
- `MEarnerManager` is meant to have a relatively low number of holders, so that `claimFor()` can realistically be called on all of them.
- `updateIndex()` will be called promptly after an update of the `MToken.earnerRate()`.
- The earner rate oracle on spoke chains is assumed to always return the `MToken.earnerRate()` from Mainnet within an acceptable time delay.

- The $M governance will not add arbitrary/malicious contracts to the earners list, nor will it altogether disable the list (making everyone an earner).
- The $M governance will not remove legitimate extensions from the earners list, as this will prevent their users from unwrapping and exiting the system (since the SwapFacility will no longer recognize them as extensions).

3 Limitations and use of report

Security assessments cannot uncover all existing vulnerabilities; even an assessment in which no vulnerabilities are found is not a guarantee of a secure system. However, code assessments enable the discovery of vulnerabilities that were overlooked during development and areas where additional security measures are necessary. In most cases, applications are either fully protected against a certain type of attack, or they are completely unprotected against it. Some of the issues may affect the entire application, while some lack protection only in certain areas. This is why we carry out a source code assessment aimed at determining all locations that need to be fixed. Within the customer-determined time frame, ChainSecurity has performed an assessment in order to discover as many vulnerabilities as possible.

The focus of our assessment was limited to the code parts defined in the engagement letter. We assessed whether the project follows the provided specifications. These assessments are based on the provided threat model and trust assumptions. We draw attention to the fact that due to inherent limitations in any software development process and software product, an inherent risk exists that even major failures or malfunctions can remain undetected. Further uncertainties exist in any software product or application used during the development, which itself cannot be free from any error or failures. These preconditions can have an impact on the system's code and/or functions and/or operation. We did not assess the underlying third-party infrastructure which adds further inherent risks as we rely on the correct execution of the included third-party technology stack itself. Report readers should also take into account that over the life cycle of any software, changes to the product itself or to the environment in which it is operated can have an impact leading to operational behaviors other than those initially determined in the business specification.

4 Terminology

For the purpose of this assessment, we adopt the following terminology. To classify the severity of our findings, we determine the likelihood and impact (according to the CVSS risk rating methodology).

- *Likelihood* represents the likelihood of a finding to be triggered or exploited in practice
- *Impact* specifies the technical and business-related consequences of a finding
- *Severity* is derived based on the likelihood and the impact

We categorize the findings into four distinct categories, depending on their severity. These severities are derived from the likelihood and the impact using the following table, following a standard risk assessment procedure.

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

As seen in the table above, findings that have both a high likelihood and a high impact are classified as critical. Intuitively, such findings are likely to be triggered and cause significant disruption. Overall, the severity correlates with the associated risk. However, every finding's risk should always be closely checked, regardless of severity.

5 Open Findings

In this section, we describe any open findings. Findings that have been resolved have been moved to the Resolved Findings section. The findings are split into these different categories:

- Design: Architectural shortcomings and design inefficiencies
- Correctness: Mismatches between specification and implementation

Below we provide a numerical overview of the identified findings, split up by their severity.

Critical-Severity Findings	0
High-Severity Findings	0
Medium-Severity Findings	1

- Account Blacklist Check Before Wrap Can Be Circumvented Acknowledged

Low-Severity Findings	3

- Discrepancy of Yield When Earning Is Disabled Acknowledged
- External Instead of Public Functions Acknowledged
- MEarnerManager Fee Recipient Can Have a Fee Rate Acknowledged

5.1 Account Blacklist Check Before Wrap Can Be Circumvented

Design Medium Version 1 Acknowledged

CS-MEXT-002

In `YieldToOne`, the hook `_beforeWrap()` checks that the source of the MToken `account` and the recipient of the wrapped token `recipient` are both not blacklisted. However, if `account` is blacklisted it is easy for them to transfer the MToken to another address and then wrap from there, circumventing the check.

Acknowledged:

M^ZERO Labs is aware of the issue but decided not to change the code.

5.2 Discrepancy of Yield When Earning Is Disabled

Correctness Low Version 1 Acknowledged

CS-MEXT-005

If the governance removes the `MYieldFee` wrapper contract from the earners list, anyone can trigger a call to stop the wrapper from earning more yield in `M` token in two ways:

1. Call `disableEarning()` in the `MYieldFee` wrapper contracts.

2. Call `stopEarning(wrapperAddr)` in `M` token contract.

The first option is the correct way to stop the wrapper earning more yield and ensure that the accounting in the wrapper is correct. However, since anyone can call `stopEarning()` in the `M` token contract, the second option is also possible. In this case, the accounting of the wrapper will be off depending on the delay that `disableEarning()` is executed. Theoretically, any delay creates solvency issues for the wrapper as the yield distributed to `MYieldFee` holders is larger than the yield earned by the wrapper, hence last users cannot unwrap their tokens.

Acknowledged:

M^ZERO Labs is aware of a potential discrepancy in the yield and solvency issues in case of a delay to stop the yield accrual in the wrapper contract.

5.3 External Instead of Public Functions

Design | Low | Version 1 | Acknowledged

CS-MEXT-006

For the sake of code readability and maintainability, functions that are not meant to be called from within the contract should have an `external` visibility. Below is a list of `public` function that can be external:

- `MEarnerManager.initialize()`
- `MYieldToOne.initialize()`
- `MSpokeYieldFee.initialize()`

Acknowledged:

M^ZERO Labs is aware of this and chose to leave the code unmodified in order to allow other contracts to build on top and call `super.initialize()`.

5.4 `MEarnerManager` Fee Recipient Can Have a Fee Rate

Design | Low | Version 1 | Acknowledged

CS-MEXT-007

The specifications of `MEarnerManager` indicate that the fee recipient should have a `feeRate` of zero. However, it is still possible for the `EARNER_MANAGER_ROLE` to call `setAccountInfo()` for the current fee recipient and apply a non-zero fee rate, or even remove it from the whitelist. In practice this should not happen as the `EARNER_MANAGER_ROLE` is a trusted role, but it is still a theoretical inconsistency with the specs.

Acknowledged:

M^ZERO Labs responded:

```
This will remain at the discretion of the earner manager role
```

6 Resolved Findings

Here, we list findings that have been resolved during the course of the engagement. Their categories are explained in the Open Findings section.

Below we provide a numerical overview of the identified findings, split up by their severity.

Critical-Severity Findings	0
High-Severity Findings	1
Medium-Severity Findings	4
Low-Severity Findings	2
Informational Findings	6

- MYieldFee Can Be Forced Into Insolvency if Non Earning Code Corrected

- Wrapped $M V1 Is Unsupported Code Corrected
- Wrong Remaining Balance in swapOut() Code Corrected
- MEarnerManager Can Be Forced Into an Irrecoverable State Code Corrected
- Removing MEarnerManager From the Earner List Will Make It Insolvent Code Corrected

- Disable Initializers on Implementation Code Corrected
- Incorrect earnerRate() Returned by MYieldFee When Not Earning Code Corrected

- Config Should Be Checked on Spoke Chains After MSpokeYieldFee Deployment Code Corrected
- Discrepancy in Event Emission Code Corrected
- Potentially Redundant Balance Difference Code Corrected
- Redundant Balance Difference Code Corrected
- Unused Code Code Corrected
- Wrong Storage Location for EIP7201 Code Corrected

6.1 `MYieldFee` Can Be Forced Into Insolvency if Non Earning

Design High Version 1 Code Corrected

CS-MEXT-001

The `MYieldFee` contract has an internal index to track its yield and fee to be distributed and relies on the value of `latestRate` to know whether it is currently earning or not. The internal index needs to be paused whenever the contract is removed from the earner whitelist. If it is not paused, the internal accounting will still account for some yield which is not received by the contract, making it insolvent.

Because `updateIndex()` is not aware of the current earning status of the contract, the index can be "unpaused" and thus force the contract to become insolvent. The following flow would reactivate the growth of the index even though `MYieldFee` is not earning on the `MToken` contract:

1. `MYieldFee` is earning on `MToken` and earning is enabled internally

2. Governance vote removes `MYieldFee` from the earners list and `MYieldFee.disabledEarning()` is called, setting the `latestRate` to 0.
3. `MYieldFee.updateIndex()` is called and sets back the `latestRate` to whatever `earnerRate()` returns, this value will be non-zero as long as the fee is not 100% or the `MToken` earner rate is non-zero.

Code corrected:

The contract now relies on a new storage variable `isEarningEnabled` to track its earning state. If this variable is not set, `updateIndex()` will not do any state changes and `currentIndex` will return the value of the index when earning was stopped on the contract.

6.2 Wrapped $M V1 Is Unsupported

Correctness | Medium | Version 2 | Code Corrected

CS-MEXT-021

The functions of the `SwapFacility` and the `UniswapV3SwapAdapter` work with exact token amounts: in particular, they expect `wrap()` and `unwrap()` on the `Wrapped $M` token to mint/burn exactly the specified number of tokens (as is the case for the extensions). However, the currently deployed version of `Wrapped $M` (Version 1) does not respect this assumption, as it incurs rounding errors upon `wrap()` and `unwrap()`. This will cause all functions on the `UniswapV3SwapAdapter` and most functions on the `SwapFacility` to revert.

Code corrected:

The contracts now use balance differences, in order to account for imprecise `wrap()/unwrap()` on Wrapped `$M` V1. A comment specifies that, once Wrapped `$M` is upgraded to V2, these balance differences will be unnecessary.

6.3 Wrong Remaining Balance in `swapOut()`

Correctness | Medium | Version 2 | Code Corrected

CS-MEXT-018

The function `UniswapV3SwapAdapter.swapOut()` is used to swap some `MExtension` tokens to arbitrary tokens. To do so, it will first swap the input token to `wMToken` if it is not already the case, and then perform the swap to the output token. It can happen that the swap router does not fully consume the input amount and some `wMToken` stays in the `UniswapV3SwapAdapter`.

This leftover amount is incorrectly tracked as the `wrappedMBalanceBefore` is computed after the swap, leading to `remainingBalance` being always `0`.

Code corrected:

The Wrapped `$M` balance is now gauged at the beginning of the function

6.4 `MEarnerManager` Can Be Forced Into an Irrecoverable State

Design | Medium | Version 2 | Code Corrected

CS-MEXT-019

The `MEarnerManager` contract relies on two variables to determine its earning state: `wasEarningEnabled` and `disableIndex`. They form a state machine with 4 states (both variables can be 0 or non-zero). Only three of these states are valid, and the only transitions that can/should happen are `(0,0) -> (!0,0) -> (!0,!0)`. The `(0,!0)` state is invalid; however, it is reachable. If, at deployment time, the contract is not yet an allowed earner, anyone can call `disableEarning()`, which will irreversibly brick its interest accrual: even calling `enableEarning()` afterwards will not remedy this.

Code corrected:

The function `isEarningEnabled()` is used to check whether the contract has earning currently enabled instead of simply `disableIndex != 0`.

6.5 Removing `MEarnerManager` From the Earner List Will Make It Insolvent

Design | Medium | Version 1 | Code Corrected

CS-MEXT-003

If an `MEarnerManager` was to be removed from and added again to the earner list, the contract would become insolvent and trigger a bank run because the yield kept accruing in the internal accounting while the contract was not actively earning. If `MEarnerManager` was to be removed from the earner list, an upgrade of the contract would be needed prior to the removal to handle it gracefully.

Code corrected:

If removed from the earner list, `disableEarning()` can be called and will set the `disableIndex` to the current value of the index. Once this is done, the contract will never accrue interest again as the index will be fixed at that value.

6.6 Disable Initializers on Implementation

Design | Low | Version 1 | Code Corrected

CS-MEXT-004

All the wrapper extensions are behind a proxy contract and thus implement an initializer function. These initialize functions are permissionless but can only be executed once, until the initialized flag is set. Note that they can also be called on the implementation contract. While this does not have any adverse effect, it is generally recommended not to leave the implementation contract uninitialized. It is recommended to invoke `_disableInitializers()` in the constructor of the implementation to automatically lock the initialize functions in the implementation contract.

Code corrected:

The initializers are disabled on the implementations of the wrapper contract through a call to `_disableInitializers()` from the constructor of `MExtenstion`, which is inherited by all the wrapper extensions.

6.7 Incorrect `earnerRate()` Returned by `MYieldFee` When Not Earning

Design | Low | Version 1 | Code Corrected

CS-MEXT-020

The function `MYieldFee.earnerRate()` does not return `0` if the contract is in non-earning mode. Instead, it keeps returning the discounted earner rate as usual.

Code corrected:

`MYieldFee.earnerRate()` returns `0` when earning is disabled on the contract.

6.8 Config Should Be Checked on Spoke Chains After `MSpokeYieldFee` Deployment

Informational | Version 1 | Code Corrected

CS-MEXT-009

The contract `MSpokeYieldFee` exposes two very similar `initialize()` functions:

- from `MYieldFee` with 9 parameters
- from `MSpokeYieldFee` with 10 parameters, basically `MYieldFee.initialize()` + 1 address

After deployment and initializations, the configuration of `MSpokeYieldFee` should be double-checked to ensure the correct initialization function was called (`MSpokeYieldFee.initialize`).

Code corrected:

The `initialize()` function of `MSpokeYieldFee` has now the same signature as `MYieldFee.initialize()` and simply overrides it.

6.9 Discrepancy in Event Emission

Informational | Version 1 | Code Corrected

CS-MEXT-010

In most of the codebase, events are not emitted if a storage variable is about to be updated to the same value is already has. The function `UniswapV3SwapAdapter._whitelistToken()` emits the event `TokenWhitelisted` even if the `isWhitelisted` will not change the value of `whitelistedTokens[token]`, which differs from the rest of the codebase.

Code corrected:

The function now returns early, in case of a no-op.

6.10 Potentially Redundant Balance Difference

Informational | Version 1 | Code Corrected

CS-MEXT-011

The function `SwapFacility.swapOutToken()` performs a `_swap(extensionIn, baseToken, amountIn, address(this))`, surrounded by a balance difference, so as to gauge the actual amount of `baseToken` received after the internal call to `wrap()`. This is currently needed, as the deployed Version 1 of the `Wrapped $M Token` has rounding errors on `wrap()`. Should the token be upgraded to another version that doesn't suffer from the same shortcomings, this balance difference would become redundant.

Code corrected

The balance difference has been removed, but this does not cause issues as long as the `Wrapped $M` token gets upgraded to a new implementation that does not incur rounding errors. See Wrapped $M V1 is unsupported.

6.11 Redundant Balance Difference

Informational | Version 1 | Code Corrected

CS-MEXT-012

The functions `_swap()` and `_swapOutM()` of `SwapFacility` recompute the `amount` parameter mid-way as a balance difference, to account for internal rounding errors in the `M` token contract. However, no rounding error is actually incurred, since the `SwapFacility` is not an earner. The re-computation of the `amount` is therefore unnecessary.

Code corrected:

The balance difference has been removed

6.12 Unused Code

Informational | Version 1 | Code Corrected

CS-MEXT-013

For the sake of code readability and maintainability, unused code should be removed from the codebase:

- the events `IUniswapV3SwapAdapter.SwappedIn` and `IUniswapV3SwapAdapter.SwappedOut` are never used

Code corrected:

The events are now emitted

6.13 Wrong Storage Location for EIP7201

Informational | Version 1 | Code Corrected

CS-MEXT-014

The `MSpokeYieldFee` contract implements the EIP-7201 for namespaced storage layout. In the `@custom` comment, it indicates the location `M0.storage.SpokeMYieldFee`, but computes the location for `M0.storage.MSpokeYieldFee` instead.

Code corrected:

The code in question was removed from the codebase.

7 Informational

We utilize this section to point out informational findings that are less severe than issues. These informational issues allow us to point out more theoretical findings. Their explanation hopefully improves the overall understanding of the project's security. Furthermore, we point out findings which are unrelated to security.

7.1 Blacklist Can Theoretically Be Evaded

Informational Version 1 Acknowledged

CS-MEXT-008

The `MYieldToOne` tokens implements a blacklist and the EIP-3009 (Transfer with authorization). The blacklist functionality won't allow a blacklisted user to spend its token allowance, but won't prevent them from using EIP-3009 signatures because they can be consumed from a fresh address that will not be blacklisted.

As a hypothetical scenario, consider a lending protocol that allows the use of EIP-3009 for the transfer of liquidity from arbitrary sources, a blacklisted account can use the EIP-3009 signature of a non-blacklisted account in order to fund its position, using it in a similar way as an allowance. To the best of our knowledge, no such protocol exists at the time of writing.

Acknowledged

M^ZERO Labs is aware of this behavior and states:

```
Since EIP-3009 is not widely used across DeFi, we won't remediate this issue.
```

7.2 Wrong `feeRate` in Event for De-Whitelisted Addresses

Informational Version 1 Acknowledged

CS-MEXT-015

In `MEarnerManager`, when an address is de-whitelisted, the function parameter for `feeRate` must be 0, but the effective fee rate from there will be 100%. The emitted event will have `status=false` and `feeRate=0`, observers must also take the `status` into account as relying only on the `feeRate` will not give the new applied fee in this case.

Acknowledged

M^ZERO Labs is aware of this behavior.

7.3 `MEarnerManager` Fee Can Round to 0

Informational Version 1 Acknowledged

CS-MEXT-016

The $M token only has 6 decimals, as do the Wrapped $M Token and all the extensions in scope of this review. Therefore, 1 wei of $M has non-negligible value, especially if taken as a per-block rate. As a reference number, 1 wei per block (12 seconds) equals $2.62 per year.
The fee in MEarnerManager is taken as a cut on the yield realized through a call to claimFor(). For small yearly amounts, this can get rounded down to 0 at every block, should the yield be claimed this frequently.
As an intentional attack, this is unlikely to be profitable, as the associated gas costs would likely exceed the 10^{-6} dollars "saved" in fees, making it a costly griefing vector.
It can also arise from mismanagement, should an admin decide to automatically sweep all the due fees at every block.

Acknowledged:

M^ZERO Labs is aware of the potential issue and decided not to change the code.

7.4 disableEarning() on Base MExtension Might Revert

Informational | Version 1 | Acknowledged

CS-MEXT-017

The function MExtension.disableEarning() first checks that earning is not already disabled on the MToken and reverts if it is the case. In the event where the extension is removed from the earner list, there are two ways to disable earning:

1. Call disableEarning() in the extension wrapper contracts.
2. Call stopEarning(wrapperAddr) in M token contract.

For the currently implemented wrappers, the only effect of this is that the event EarningDisabled will not be emitted from the wrapper contract, but this behavior should be kept in mind when developing new wrapper extensions.

This does not apply to MYieldFee as it implements a different way to check whether earning is enabled.

Acknowledged

M^ZERO Labs is aware of this issue and states:

```
We will make sure to keep this behavior in mind when developing new extensions and communicate this mechanism to potential integrators.
```

8 Notes

We leverage this section to highlight further findings that are not necessarily issues. The mentioned topics serve to clarify or support the report, but do not require an immediate modification inside the project. Instead, they should raise awareness in order to improve the overall understanding.

8.1 Claiming Less Often Results in More Yield

Note **Version 1**

In `MEarnerManager`, claiming yield reduces one's principal (since the fee is transferred out to the `feeRecipient` using `_update()`). This reduces the yield generated afterwards, compared to somebody who claims less often, thus "holding onto" their principal for a longer period of time.
In the limit case where claiming is "continuous", the resulting balance curve is an exponential whose rate is equal to `earnerRate() * (1 - feeRate)` (exactly as in `MSpokeYieldFee`). In the opposite limit case, claiming only once results in a "sampled" exponential, whose rate is the full `earnerRate()`, but whose final result is scaled down by `(1 - feeRate)`: this is strictly higher.

8.2 Contracts Can Be Temporarily Insolvent

Note **Version 1**

Due to the rebasing nature of the `MToken`, the wrapper contracts may be temporarily insolvent. In practice, this should not impact security or UX as the rounding error should balance itself over time with users wrapping and unwrapping, and the yield accumulated should be able to cover for this small difference. However, it is important to note that this is a potential risk that should be monitored.

8.3 Effects of Disabling the $M Earners List

Note **Version 1**

The `SwapFacility` checks whether an address is an approved extension just by checking whether it is an `$M` earner. If the governance were to ever disable the earners list, thus making everyone an earner, this would open the door to phishing attacks: anyone would be able to operate a malicious smart contract that just exposes the `wrap()` and `unwrap()` functions, and lure people into depositing `$M` tokens through the legitimate `SwapFacility`.
Notice that this can also happen if the governance explicitly adds such a malicious contract to the earners list, or if an earner contract gets upgraded to a malicious implementation.

8.4 Exact Token Amounts

Note **Version 3**

As of this writing, the currently deployed version of Wrapped `$M` (Version 1) incurs rounding errors on `wrap()` and `unwrap()`. This affects all functions of the `SwapFacility` and the `UniswapV3SwapAdaper`, except for `UniswapV3SwapAdaper.swapOut()`. Until Wrapped `$M` gets upgraded to a new implementation that does not suffer from rounding errors, callers of these functions should not expect the balance differences to be exactly equal to the parameters they specified.

8.5 Extensions May Not Be Solvent to the Last Wei

Note **Version 1**

The extension contracts are assumed to be in earning state (most of the time). If this is the case, the `unwrap()` function will trigger an `$M` transfer that will *round up* the principal amount to be deducted from the contract's `rawBalance`. If all users of an extension were to claim their due yield and then `unwrap()`, these roundings may add up to the point where the last operation fails and reverts with `InsufficientBalance`, because the principal amount to be deducted is greater than the leftover `rawBalance` of the extension contract by a few weis.
It is therefore to be noted that claiming all yield and then unwrapping the whole balance might not always succeed. Users are discouraged from operating smart contracts that can only exit the system in this way, and instead allow for more flexibility (e.g. specify an amount to `unwrap()`).

8.6 Freezelist Front-running

Note **Version 1**

If the transaction freezing a user is published in a public mempool, the user might be able to avoid being frozen by front-running the transaction, transferring their funds to a new address.

Developers should be aware of this possibility when implementing block lists for compliance purposes.

8.7 Transfer Events Can Have De-Whitelisted Addresses as Sender

Note **Version 1**

In `MEarnerManager`, a whitelisted address holding funds can be removed from the whitelist. Such an address should not be able to be the origin or recipient of a token transfer. Even though their assets are not seizable, 100% of the yield generated can be claimed. When `claimFor()` is used, a `Transfer` event will be emitted with the de-whitelisted address as the sender.

8.8 Updates of the Effective Earner Rate on L2s

Note **Version 1**

The "effective earner rate" of `MSpokeYieldFee` depends on three parameters:

1. The protocol's earner rate, supplied by the oracle
2. The `feeRate`, set by admins
3. The contract's earning status (if non-earner, the rate goes to 0)

When either of these change, a call to `MSpokeYieldFee.updateIndex()` should be performed, to correctly record the new rate into the storage variable `latestRate`.
However, this function call only brings `latestUpdateTimestamp` up to `IContinuousIndexing(mToken()).latestUpdateTimestamp()`, not to the present `block.timestamp`. Therefore, the new rate will take effect retroactively, unless `M.updateIndex()` gets called (by the portal) right before `MSpokeYieldFee.updateIndex()`.

8.9 Yield From `MYieldFee` Can Be Taken Away From Users

Note Version 1

Even though the role is trusted to not act maliciously, users must be aware that the `claimRecipientManager` address of `MYieldFee` has the power to set a claim recipient for arbitrary `MYieldFee` holders, effectively taking their yield.

8.10 `projectedTotalSupply()` Is an Approximation

Note Version 1

The specification of the function `projectedTotalSupply()` in `MEarnerManager` and `MYieldFee` describe the resulting value as:

```
The projected total supply if all accrued yield was claimed at this moment.
```

But this value is actually an approximation (upper-bound) of the total supply if all actors were to claim the yield, as the claimed yield is rounded down for each of them.