PUBLIC

Code Assessment

of the Makina Periphery Smart Contracts

January 9, 2026

Produced for

MAKINA

by

CHAINSECURITY

Contents

1 Executive Summary

Dear Makina Team,

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

Makina implements periphery contracts for the Makina Protocol, an on-chain asset management platform. They include a Depositor, a Redeemer with a withdrawal delay, and a Security Module.

The most critical subjects covered in our audit are access control, functional correctness, and fee management. Security regarding Access control is high. Security regarding functional correctness is high, after StartCooldown Allows Starting Cooldown With More Than Balance has been resolved. Security regarding fee management is high, after Watermark Does Not Consider Maximum Fee was fixed.

The general subjects covered are code complexity, upgradeability, and documentation. Security regarding all the aforementioned subjects is high.

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	0
Medium-Severity Findings	3
• Code Corrected	2
• Acknowledged	1
Low-Severity Findings	2
• Code Corrected	1
• Risk Accepted	1

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 Makina Periphery 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	25 August 2025	cee5437d5a7fbb3b843a3498ed5ac164429a82f6	Initial Version
2	12 September 2025	bffaeabeb8208f93dc765202dde3b3d3ca791c67	Fixes
3	09 December 2025	c8b8df1aaff00053a94ce395e593cc97957b5b4f	Helpers & Minor Changes
4	09 January 2026	e8b2b2411f6e534177e79953d4414e8369c7d524	Fixes

For the Solidity smart contracts, the compiler version `0.8.28` was chosen.

As part of version 1, the following contracts are included in scope for the Periphery Repository:

```
depositors/
    DirectDepositor.sol
factories/
    HubPeripheryFactory.sol
fee-managers/
    WatermarkFeeManager.sol
flashloans/
    FlashloanAggregator.sol
redeemers/
    AsyncRedeemer.sol
registries/
    HubPeripheryRegistry.sol
security-module/
    SecurityModule.sol
utils/
    MachinePeriphery.sol
    MakinaPeripheryContext.sol
    Whitelist.sol
```

Note that FlashloanAggregator was initially audited as part of a separate audit for the core of the Makina Protocol, and the findings from that audit are not included in this report.

In Version 2, the following contracts were added to the scope of the assessment:

```
factories/
    MetaMorphoOracleFactory.sol
oracles/
    ERC4626Oracle.sol
security-module/
    SMCooldownReceipt.sol
```

In Version 3, the following contracts were added to the scope of the assessment:

```
weiroll-helpers/
    BooleanHelper.sol
    Bytes32Helper.sol
    CastHelper.sol
    ContextHelper.sol
    KeyValueStore.sol
    MathHelper.sol
    SignedMathHelper.sol
```

2.1.1 Excluded from scope

Any file not explicitly listed in the Scope section is out of scope. In particular, external libraries (e.g. OpenZeppelin), deployment scripts, and tests are excluded.

Additionally, all configurations and operational usage are out of scope, including the management, role assignments, registry contents, and the chosen cooldowns, limits, or risk thresholds.

2.2 System Overview

This system overview describes the updated version (Version 2) of the contracts as defined in the Assessment Overview.

At the end of this report section, we have added subsections for each of the changes corresponding to the versions.

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

Makina implements a periphery layer to complement the core of the protocol. The periphery contracts provide user-facing modules for deposits, redemptions, fee distribution, and a security module. There is also a factory/registry to deploy them.

2.2.1 Periphery Base & Context

2.2.1.1 MakinaPeripheryContext & MachinePeriphery

All periphery modules extend two base contracts:

- MakinaPeripheryContext, which exposes an immutable `peripheryRegistry` (a directory of beacons and the active factory).
- MachinePeriphery, which binds the module to one Machine with a factory-only, one-time `setMachine` function. The MachinePeriphery role-check modifiers (`onlyMechanic`,

`onlySecurityCouncil`, `onlyRiskManager` and `onlyRiskManagerTimelock`) read from the bound Machine directly.

2.2.1.2 Whitelist

Periphery contracts can additionally extend an AccessManager-controlled allowlist. This abstract contract primarily provides the `whitelistCheck` modifier for gating entry points, and the view functions `isWhitelistEnabled` and `isWhitelistedUser`.

The contract provides two administration functions. The `setWhitelistStatus` function enables or disables the whitelist globally, and the `setWhitelistedUsers` function adds or removes users. They are both callable only by the Risk Manager.

2.2.2 Registries & Factory

2.2.2.1 HubPeripheryRegistry

The HubPeripheryRegistry is the on-chain directory of periphery beacons and the active factory.

The registry maintains beacons keyed by an implementation ID. The implementation ID (a `uint16`) is a selector for a concrete implementation line within a periphery type. For example, an ID can point to a specific Redeemer implementation such as AsyncRedeemer. The security module is the exception and has a single beacon without an ID key. The beacons themselves point to the actual implementation, which can be upgraded using `upgradeTo`.

The registry exposes the view functions `peripheryFactory`, `depositorBeacon`, `redeemerBeacon`, `feeManagerBeacon`, and `securityModuleBeacon` to read the beacons.

The `restricted` admin functions are `setPeripheryFactory`, `setDepositorBeacon`, `setRedeemerBeacon`, `setFeeManagerBeacon`, and `setSecurityModuleBeacon`.

The Authority is governed by the registry's own AccessManager set at construction.

2.2.2.2 HubPeripheryFactory

The HubPeripheryFactory is the deployment and configuration hub referenced by the registry. It deploys periphery instances as `BeaconProxy` contracts from the beacons registered in the registry and configures them with a specific Machine.

Depositors, redeemers, and fee managers are deployed using `createDepositor`, `createRedeemer`, and `createFeeManager`. Creation takes an implementation ID to choose the beacon and records the ID for the new instance. The `isDepositor`, `isRedeemer`, and `isFeeManager` functions allow checking if an address was created by the factory and of which type. The `depositorImplemId`, `redeemerImplemId`, and `feeManagerImplemId` functions return the implementation ID for a given instance (reverting if not of that type).

The security module is deployed from its single beacon via `createSecurityModule`. The `isSecurityModule` function checks if an address was created by the factory as a security module.

After creation, the `setMachine` function allows configuring an instance of a periphery contract with a Machine. The `setSecurityModule` function configures a fee manager with a security module.

2.2.3 Periphery Components

2.2.3.1 DirectDepositor

The DirectDepositor is a minimal adapter that forwards accounting-token deposits to the Machine. It extends the Whitelist and, as with all periphery contracts, it extends MachinePeriphery and MakinaPeripheryContext.

The DirectDepositor offers a single function: `deposit`, gated by a `whitelistCheck`. It pulls the tokens from the `msg.sender` and calls `Machine.deposit()`. The `minShares` parameter, for which the user is responsible, provides slippage protection.

2.2.3.2 AsyncRedeemer

The AsyncRedeemer is a queue-based redemption module that uses ERC-721 NFTs. It extends MachinePeriphery, MakinaPeripheryContext, and the Whitelist.

Users create requests with the `whitelistCheck`-protected function `requestRedeem`. The function pulls Machine share tokens from the `msg.sender` and mints a queue NFT to the `recipient`. At the same time, it records the value of the share tokens in accounting tokens. After a `finalizationDelay`, the mechanic batches redemptions with `finalizeRequests()`. For each request, the contract uses the minimum of the current conversion rate and the value recorded at request time. It calls `Machine.redeem()` and marks the range as finalized. Finally, holders can claim with `claimAssets`, which burns the NFT and transfers accounting tokens.

The delay is updated using `setFinalizationDelay()` by the Risk Manager, and all user functions are gated by `whitelistCheck`.

2.2.3.3 Watermark Fee Manager

The Watermark Fee Manager is the fee engine consulted by the Machine during `updateTotalAum()`. When called, the Machine checks the fee mint cooldown. If enough time has elapsed, it asks the manager for a fixed fee via `calculateFixedFee` and a performance fee via `calculatePerformanceFee`. The Machine then mints the total fee in share tokens and calls `distributeFees()` on the manager.

Fixed fee represents continuous management charges and is a per-second rate of the total share supply. Note that this value is computed continuously but minted discretely. As such, the per-second rate is applied as if the total share supply were constant since the last fee computed. The fixed fee rate is itself the sum of two parts: a portion that goes to the Security Module (if set) and a portion that goes to other receivers.

The performance fee is charged on a percentage of the positive share-price increase above the watermark. The manager tracks a share-price watermark: on the first call, it initializes the watermark; afterwards, a performance fee is returned only when the new price is strictly above the stored watermark, and the watermark is updated to that new price. Note that the WatermarkFeeManager uses an `adjustedSharePrice` which is computed by the Machine using the current AUM and a supply that includes the fixed fee. This neutralizes the fixed-fee dilution in the performance calculation. The manager stores this `adjustedSharePrice` as the watermark without including the dilution later caused by the minted performance fee. This favors the Machine shareholders, as the watermark will be higher than if the dilution were included.

The Machine enforces a hard cap on performance and management fees per second. If either of the fees exceed the cap, the Machine reduces the corresponding fee to the cap. It then mints the total fee in share tokens to itself, approves the manager for exactly that amount, and calls `distributeFees()`. The manager pulls shares from the Machine and splits them according to configured receivers and BPS splits. An optional slice of the fixed fee can be routed to the Security Module if set. After distribution, the Machine clears the approval and burns any dust left on itself (e.g., from rounding).

Rates, receiver lists, and BPS splits are `restricted` and governed by the Machine's AccessManager through the manager's `authority()` override. The watermark can be reset to the current price (downward only). The `setSecurityModule` function can be called by the factory exactly once as part of the deployment process to configure the security module. The security module must belong to the same Machine.

2.2.3.4 Security Module

The Security Module provides a supply of Machine shares that can be burned. Burning reduces the circulating share supply, and increases the price of the remaining Machine shares. Users who opt in to bear this risk lock their Machine shares in the module and hold the security module token (`sm-<SYMBOL>`). In return, their position grows as the Fee Manager transfers newly minted shares to the module, increasing the pool's assets per `sm` token.

Locking and redemptions are pool-based. Users deposit machine share tokens with `lock()` and receive `sm` tokens proportional to the pool ratio. They can redeem by first entering a cooldown period with `startCooldown()` and then calling `redeem()` after the cooldown has passed. Redemptions pay the minimum of the current pool conversion and the snapshot taken at cooldown start time. Conversions use the standard OpenZeppelin ERC4626 pool math:

```
shares = assets * (totalSupply + 1) / (totalLocked + 1)
assets = shares * (totalLocked + 1) / (totalSupply + 1)
```

Slashing is controlled by the Security Council. `slash()` burns Machine shares from the module up to `maxSlashable()`, which is bounded by both a cap in basis points (`maxSlashableBps`) and a floor on residual balance (`minBalanceAfterSlash`). Note that the `maxSlashableBps` is per slashing call. If `slash()` is called multiple times, the maximum can be reached each time. Entering slashing sets `slashingMode = true` and blocks new locks. `settleSlashing()` clears this state. Redemptions remain available.

Risk parameters (`cooldownDuration`, `maxSlashableBps`, and `minBalanceAfterSlash`) are adjustable by the Risk Manager.

2.2.3.5 Changes in Version 2

The following changes were made in Version 2 of the codebase:

- **Permissions:** The time-locked risk manager role was introduced by adding `onlyRiskManagerTimelock` to MachinePeriphery. `AsyncRedeemer.setFinalizationDelay` and the Security Module setters (`setCooldownDuration`, `setMaxSlashableBps`, `setMinBalanceAfterSlash`) now use `onlyRiskManagerTimelock`.
- **Whitelist**: The whitelist base contract no longer exposes public administration functions. `DirectDepositor` and `AsyncRedeemer` now expose `setWhitelistStatus` / `setWhitelistedUsers` guarded by `onlyRiskManager`.
- **Security Module:**
  - The SecurityModule's cooldown functionality was reworked, and an ERC-721 NFT (`SMCooldownReceipt`) contract was introduced to represent cooldowns.
  - `startCooldown()` now mints a receipt NFT and transfers the shares that are in cooldown to the SecurityModule.
  - `cancelCooldown()` can now be used to cancel a cooldown, but only if the cooldown is not already complete. It returns the shares to the user and burns the receipt NFT.

The following new components were added in Version 2 of the codebase:

- **Morpho Oracles:**
  - **ERC4626Oracle:** Reports the price of one vault share (e.g., a Morpho Vault) in units of its underlying token and exposes it through `AggregatorV2V3Interface`. Consumers can read it like a Chainlink feed. `latestAnswer()` / `latestRoundData()` return the on-chain exchange rate computed at read time as `vault.convertToAssets(ONE_SHARE)`, scaled to the feed's chosen `decimals`. The feed uses a constant round id `1` and sets both `startedAt` and `updatedAt` to `block.timestamp`. There is no heartbeat, caching, or historical rounds.

- **MetaMorphoOracleFactory:** Lets governance create ERC4626Oracle feeds. Only vaults recognized as MetaMorpho by an allow-listed Morpho factory can be used. Oracles are deployed via the restricted function `createMetaMorphoOracle` function, which requires the `factory` to be whitelisted with `setMorphoFactory()` and then verifies the vault with `IMetaMorphoFactory.isMetaMorpho()`. On success, it deploys a new immutable `ERC4626Oracle` and records it, so that `isOracle()` can attest provenance. The access control is done using OpenZeppelin's `AccessManagedUpgradeable` with an initializer to set the authority.

2.2.3.6 Changes in Version 3

The following changes were made in Version 3 of the codebase:

- **DirectDepositor:** An optional `referralKey` argument was added to `deposit` and forwards it to `Machine.deposit`.
- **AsyncRedeemer:**
  - A global `minRedeemAmount` was added. `requestRedeem()` reverts if the share amount provided is below the threshold. The value can be updated via `setMinRedeemAmount` by the Timelocked Risk Manager.
  - `requestRedeem()` now takes a `minAssets` parameter.
- **Weiroll Helper Libraries:** Helper contracts (`BooleanHelper`, `Bytes32Helper`, `CastHelper`, `ContextHelper`, `KeyValueStore`, `MathHelper`, `SignedMathHelper`) providing math/logic/casting/context utilities and a Caliber-owned key-value store were added for weiroll programs.

2.3 Trust Model

2.3.1 Roles and Privileges

The HubPeripheryFactory and HubPeripheryRegistry receive an AccessManager at initialization.

Periphery instances enforce permissions in two ways: The Depositor, Redeemer, and Security Module use role modifiers that read roles directly from the bound Machine (onlyMechanic, onlySecurityCouncil, onlyRiskManager, onlyRiskManagerTimelock). The FeeManager uses AccessManager-governed functions. It delegates authority by implementing `authority() = IAccessManaged(machine()).authority()`. This means that its `restricted` calls are controlled by the same AccessManager as the Machine that it is used with.

2.3.1.1 Governance

This role, held by a DAO or multisig managing the OpenZeppelin AccessManager contract, has the highest level of authority. It controls all `restricted` functions. The system's security relies on a distributed trust model where different roles have specific, limited powers. However, ultimate control over the system and its assets rests with the Governance role, which is considered fully trusted.

Trust level: Fully trusted.

Capabilities: Upgrades periphery beacons, deploys and configures modules (via the factory), and configures fee manager parameters. In the worst case, Governance could maliciously upgrade contracts, leading to a total loss for users.

2.3.1.2 Risk Manager

Trust level: Partially trusted.

Capabilities: Adjusts operational risk parameters exposed by periphery modules. Examples include the AsyncRedeemer's `finalizationDelay` and the Security Module's `cooldownDuration`, `maxSlashableBps`, and `minBalanceAfterSlash`. In the worst case, the risk manager's timelock could temporarily DoS withdrawals by setting very long delays. This could be solved by governance setting a new risk manager.

2.3.1.3 Mechanic

Trust level: Partially trusted.

Capabilities: Mainly responsible for finalizing redemption batches in AsyncRedeemer. The mechanic cannot mint/burn shares arbitrarily and is bounded by per-request clamps and Machine slippage checks. Misbehavior can delay or grief, but cannot bypass those limits.

2.3.1.4 Security Council

Trust level: Partially trusted.

Capabilities: Controls slashing in the Security Module (`slash` and `settleSlashing`), bounded by `maxSlashable()`.

2.3.1.5 End Users

Trust level: Untrusted.

Capabilities: Interact with Depositor (if whitelisted), submit and claim redemptions through AsyncRedeemer, and lock/redeem Machine shares in the Security Module, all subject to module rules (cooldowns, slashing mode, slippage).

2.3.2 Upgradeability

The HubPeripheryRegistry and HubPeripheryFactory are upgradeable. Periphery instances (Depositor, Redeemer, Fee Manager, Security Module) are deployed as BeaconProxy contracts pointing to beacons registered in the registry. They can be upgraded by rotating beacon implementations.

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 errors 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

- Watermark Does Not Consider Performance Fee Dilution Acknowledged

Low-Severity Findings	1

- Fixed Fee May Be Charged on New Deposits Risk Accepted

5.1 Watermark Does Not Consider Performance Fee Dilution

Design Medium Version 1 Acknowledged

CS-MAPE-003

In WatermarkFeeManager, the watermark is set to a share price that takes into account dilution from fixed fees, but not dilution from performance fees. As a result, the watermark can be set higher than the actual share price after performance fees are minted.

Consider the following example. Assume 20% performance fee and no fixed fee.

1. User mints 100 shares for $100.
2. `updateTotalAum()` is called to set the watermark to $1.
3. Total assets are now $200. `updateTotalAum()` is called again.
4. `calculatePerformanceFee()` will calculate 10 shares fee. It will set the watermark to $2.
5. 10 shares are minted to the manager.
6. Now the share price is $200 / 110 = $1.82.

The watermark is set to $2 even though the share price after minting the fees is only $1.82. Any future profits between $1.82 and $2 will not be charged performance fees.

Acknowledged:

Makina acknowledges this issue and considers this behavior correct. Profit between the post-fee price and the pre-fee price (here, $1.82 -> $2) should not be charged performance fees again.

5.2 Fixed Fee May Be Charged on New Deposits

Design | Low | Version 1 | Risk Accepted

CS-MAPE-004

The fixed fees accrue over the elapsed time since the last fee mint. Fees are minted discretely when accounting is updated and after a cooldown. Deposits and withdrawals do not automatically mint fees.

When fees are minted, the calculation uses the current total share supply applied to the entire elapsed time window since the previous mint. As a result, any shares minted during the accrual window are included in the supply used for fee calculation, covering the full window length, even if those shares did not exist during part of that window. Conversely, shares that were redeemed before the fee mint are excluded from the supply used for that same window.

If the `feeMintCooldown` has passed, users can call `updateTotalAum()` to trigger fee minting before making a deposit. Otherwise, they will incur some management fees on their shares for the period before they deposited.

Risk accepted:

The Makina team acknowledges that the fee mechanism (minting new shares that dilute all shareholders) does not allow fees to be applied only to specific users. They state that the permissionless accounting functions (such as users being able to trigger fee minting before depositing) help mitigate this limitation.

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	0
Medium-Severity Findings	2

- StartCooldown Allows Starting Cooldown With More Than Balance Code Corrected
- Watermark Does Not Consider Maximum Fee Code Corrected

Low-Severity Findings	1

- Users Are Incentivized to Always Be in Cooldown Code Corrected

Informational Findings	2

- setMachine Checks Security Module Flag Code Corrected
- setWhitelistedUsers Emits Event Even When There Is No State Change Code Corrected

6.1 StartCooldown Allows Starting Cooldown With More Than Balance

Correctness Medium Version 1 Code Corrected

CS-MAPE-001

In SecurityModule, the `startCooldown` function accepts a `shares` amount and starts a cooldown using this amount. However, there is no check that the user has a sufficient share balance.

A user can start a cooldown for an arbitrary amount of shares, even if they do not own that many shares. In particular, a user could even start their cooldown before they have locked anything. They could also use this to start a cooldown on a separate address, then only transfer tokens to that address when they want to redeem. This effectively bypasses the cooldown mechanism.

Having an old cooldown in place could also be beneficial in case there is a slashing event. As the share price can become lower than the cooldown start price, the user could earn yield during the cooldown period even though they were not slashed.

Note that shares are transferable, so checking the balance at the time of calling `startCooldown()` is not sufficient to prevent the issue.

Code corrected:

`startCooldown` now locks the requested shares by transferring them from the caller to the module. This enforces ownership and balance at call time, removing the ability to start a cooldown without holding (or before acquiring) the shares.

`redeem` burns the locked shares from the module's balance. `cancelCooldown` was added to cancel a cooldown and return the locked shares to the requester if the cooldown has not elapsed yet.

6.2 Watermark Does Not Consider Maximum Fee

Design | Medium | Version 1 | Code Corrected

CS-MAPE-002

In MachineUtils, the `manageFees()` function enforces a maximum fee accrual rate per second. If the `totalFee` exceeds the maximum, the fees are reduced. However, if the performance fee is reduced this way, the WatermarkFeeManager will still increase the watermark by the full amount. As a result, the performance fees may depend on when exactly fees are calculated.

Consider the following example where the `feeMintCooldown` is set to 120 seconds, the `maxFeeAccrualRate` is set to 0.1 shares per second, the fixedFee is 0 and the performance fee is 20%:

1. There are 1000 shares and $1000 AUM.
2. `manageFees()` is called, updating the `_lastMintedFeesTime`. The watermark is set to $1.
3. A Caliber's reward tokens (that had accrued over a longer time) are swapped to base tokens, causing a sudden increase in AUM of $100.
4. `manageFees()` is called exactly 120 seconds after the previous call. The performance fee is calculated as 20 shares. The share price watermark is updated to $1.1. However, the maximum fee accrual rate is 12 shares (0.1 * 120), so only 12 shares are minted as fees.
5. The watermark was increased as if the full 20 shares had been minted, leading to a loss of performance fees compared to the same profit being accounted after 200 seconds instead of 120 seconds.

This issue only appears if the `maxFeeAccrualRate` is sufficiently small compared to the `feeMintCooldown`. Note that the `maxFeeAccrualRate` is a constant number of shares, not a percentage of total shares. As a result, this rate will become comparatively smaller as the number of shares increases.

A second related behavior is the following: The `calculatePerformanceFee()` call uses an `adjustedSharePrice` that assumes the `fixedFee` will be fully applied. However, if the `totalFee` caps the fees taken, the `fixedFee` used to calculate the `adjustedSharePrice` will only be partially applied. As a result, the calculated value will be inconsistent with the actual share price after the fixed fee. As the watermark is set to the `adjustedSharePrice`, this will also result in an incorrect watermark.

Code corrected:

The issue was fixed by making changes in the `makina-core` fee logic.

The single `maxFeeAccrualRate` was replaced by `maxFixedFeeAccrualRate` and `maxPerfFeeAccrualRate`. In `manageFees()`, `fixedFee` is computed and immediately capped, then used to compute the adjusted share price. `perfFee` is calculated from that adjusted price and independently capped.

Furthermore, caps are now percentages of the current supply per second, not constant shares per second. This makes it much less likely for correctly configured caps to be hit when calculating performance fees. Operators still need to ensure that both max accrual rates are set high enough relative to `_feeMintCooldown` to avoid unintended fee limiting. If the performance fee cap is reached, the WatermarkFeeManager will still set the watermark to the uncapped share price.

6.3 Users Are Incentivized to Always Be in Cooldown

Design | Low | Version 1 | Code Corrected

CS-MAPE-005

In SecurityModule, a user can start a cooldown and cancel it at any time. When a cooldown is started, the share price is recorded. When withdrawing, the share price is capped to the recorded price. However, by canceling a cooldown and starting a new one, a user can reset the recorded price to a higher value. As a result, there is no downside to being in cooldown, and users are always incentivized to be in cooldown, as it allows for more flexibility without any cost (aside from gas). Additionally, having a completed cooldown could be used to frontrun a slashing transaction if it can be predicted.

Code corrected:

Cooldown cancellation is now only allowed before cooldown maturity. Users cannot keep a finished cooldown and then cancel/restart it to reset the price snapshot at no cost. Any yield after the snapshot is not paid on redemption, creating a real opportunity cost to holding a finished cooldown and removing the incentive to always keep one.

Users can still cancel/restart an active (not yet matured) cooldown at any time. This does not allow frontrunning a price drop because the cooldown is not over. It is still possible to continuously have a partially completed cooldown and keep resetting it before completion. This could be used to keep open the option of exiting sooner than the full cooldown duration. However, it requires repeated transactions and thus incurs gas costs and management complexity.

6.4 setMachine Checks Security Module Flag

Informational | Version 1 | Code Corrected

CS-MAPE-007

`HubPeripheryFactory.setMachine()` allows the call to proceed if `machinePeriphery` is registered as depositor, redeemer, fee manager, or security module:

```
if (!_isDepositor && !_isRedeemer && !_isFeeManager && !_isSecurityModule) revert NotMachinePeriphery();
```

However, in `SecurityModule` the parent setter is explicitly disabled: `_setMachine(address)` is overridden to `revert Errors.NotImplemented()` and `machine()` is derived from the owner of `_machineShare`.

As a result, checking for `_isSecurityModule` could be removed from `setMachine()`.

Code corrected:

`HubPeripheryFactory.setMachine()` no longer accepts Security Modules. The `_isSecurityModule` allow-check was removed.

6.5 setWhitelistedUsers Emits Event Even When There Is No State Change

Informational | Version 1 | Code Corrected

In Whitelist, `setWhitelistedUsers` iterates over `users` and unconditionally assigns `_isWhitelistedUser[user] = whitelisted` for each entry. For every entry, it then emits `UserWhitelistingChanged(user, whitelisted)` regardless of whether the mapping value was already equal to `whitelisted`.

Code corrected:

`setWhitelistedUsers` now checks current state before writing and emitting.

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 All Performance Fees Suspended After Drawdown

Informational **Version 1**

CS-MAPE-006

The WatermarkFeeManager maintains a single `sharePriceWatermark`. There is no per-user watermark tracking. After a loss, the watermark remains at the prior peak. If a user deposits while the share price is below that peak, the recovery from the entry price up to the previous watermark does not accrue performance fees for that user. In other words, a depositor who joins after a drawdown benefits from "free performance" until the old watermark is reached.

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 Discrepancy Between Nominal and Effective Fees Due to Dilution

Note Version 1

When `manageFees()` is triggered by an accounting update, it first calculates a nominal fee (e.g., "20% of profit"). This fee is then paid by minting new shares. Since minting expands the total share supply while the AUM remains unchanged, the price of each share is diluted. Consequently, the newly minted fee-shares will be valued at this lower, post-dilution price. This causes the effective fee to be less than the nominal fee.

8.2 Users and Integrators Must Be Aware of Share Price Inflation

Note Version 1

The Machine (which can be deposited into with DirectDepositor), the SecurityModule and the AsyncRedeemer use shares to track user deposits. When depositing, users must use the `minShares` parameter to specify their expected share price. If they do not, there is a risk of an ERC4626 frontrunning attack, in which an attacker unexpectedly increases the share price, leading to a large rounding loss. As `balanceOf()` accounting is used, the share price can easily be increased by donations.

A similar issue can occur if an integration uses the share price to value the shares. This can be especially problematic if shares are used as collateral in a lending system.

Both users and integrating smart contracts must protect themselves from sudden share price increases.