PUBLIC

Code Assessment

of the Makina Core Smart Contracts

September 18, 2025

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 Core according to Scope to support you in forming an opinion on their security risks.

Makina implements a cross-chain asset management protocol that defines roles that can manage the assets deposited by users. The system leverages the Weiroll VM to allow the execution of arbitrary instructions that have been pre-approved by the risk manager. Furthermore, it implements slippage checks that ensure there can be no unexpectedly large losses when executing instructions or swaps. This framework allows a flexible way to manage assets across multiple chains while ensuring that losses can be limited, even in the event of an operator secret key compromise.

The most critical subjects covered in our audit are accounting correctness, reentrancy protection, and access control. Security regarding accounting correctness is high after issues such as Bridge Transfer Can Be Counted as Profit and Linked Positions Can Be Updated Separately have been addressed. Security regarding reentrancy protection is high, after issues such as Reentrancy Can Cause Incorrect Share Price and Reentrancy Can Circumvent Slippage Check have been addressed. Security regarding access control is high.

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

A correct configuration is critically important for the system to behave correctly. We have compiled a summary of important behaviors that risk managers should be aware of in the Notes for Risk Managers section.

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	2
• Code Corrected	2
Medium-Severity Findings	5
• Code Corrected	5
Low-Severity Findings	7
• Code Corrected	3
• Specification Changed	2
• Risk Accepted	1
• Acknowledged	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 Core 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	12 May 2025	d39008c7 (core) & 6c2085a9 (periphery)	Initial Version
2	30 Jun 2025	9e9f386e (core) & 0f301ca7 (periphery)	First fixes
3	16 July 2025	cf20345b (core) & 0f301ca7 (periphery)	Second fixes
4	17 Sept 2025	f293b732 (core)	Periphery Audit 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 Core Repository:

```
src/
    bridge/
        adapters/
            AcrossV3BridgeAdapter.sol
            BridgeAdapter.sol
        controller/
            BridgeController.sol
    caliber/
        Caliber.sol
        CaliberMailbox.sol
    factories/
        BridgeAdapterFactory.sol
        CaliberFactory.sol
        HubCoreFactory.sol
        SpokeCoreFactory.sol
    libraries/
        CaliberAccountingCCQ.sol
        DecimalsUtils.sol
        Errors.sol
        MachineUtils.sol
    machine/
        Machine.sol
        MachineShare.sol
    pre-deposit/
        PreDepositVault.sol
    registries/
        ChainRegistry.sol
        CoreRegistry.sol
        HubCoreRegistry.sol
        OracleRegistry.sol
```

```
        SpokeCoreRegistry.sol
        TokenRegistry.sol
    swap/
        SwapModule.sol
    utils/
        MakinaContext.sol
        MakinaGovernable.sol
src-ir/
    WeirollVM.sol
```

The following contract is included in scope for the Periphery Repository:

```
src/
     flashloans/
         FlashloanAggregator.sol
```

In Version 4, the following contract was added to the scope of the assessment for the Core Repository:

```
src/
    factories/
        Create3Factory.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. Enso-Weiroll, OpenZeppelin), bridge implementations (e.g. Wormhole, Across), deployment scripts, and tests are excluded.

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

2.2 System Overview

This system overview describes the initially received version (Version 1) 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 according to the versions.

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

Makina offers a cross-chain asset management system. It consists of two main components: the Machine and the Caliber. The Machine is responsible for managing deposits, withdrawals, and share price calculations, while the Caliber provides execution and accounting logic. The Machine is only deployed on the Hub Chain, while a Caliber is deployed on every chain (Hub and Spoke Chains).

2.2.1 MakinaGovernable

The MakinaGovernable contract serves as a governance layer from which multiple contracts inherit.

2.2.1.1 Roles

The contract defines four key governance roles:

- Mechanic: The primary operational role responsible for day-to-day system operations. In normal mode, the mechanic has operator privileges.
- Security Council: A privileged role with emergency powers. The security council can enable recovery mode and assume operator privileges when recovery mode is active.
- Risk Manager: Responsible for risk parameter management and system configuration changes that require immediate execution.
- Risk Manager Timelock: A time-delayed version of the risk manager role, used for critical system changes that require a delay period for security.

The contract defines an additional virtual role, the Operator, via a modifier `onlyOperator`. This modifier restricts access to either the mechanic (in normal mode) or the security council (in recovery mode).

Furthermore, the contract inherits from OpenZeppelin's AccessManagedUpgradeable. This means that MakinaGovernable contracts can use the `restricted` modifier to delegate the access control of a function to an external AccessManager contract that can grant and revoke permissions.

All role updates are protected by the `restricted` modifier.

2.2.1.2 Recovery Mode

The contract defines a recovery mode mechanism that can be activated by the security council. When recovery mode is enabled:

- The security council assumes operator privileges instead of the mechanic.
- Certain functions (e.g., `updateTotalAum()`, `deposit()`, and `withdraw()`) can only be called if not in recovery mode. They are protected by the `notRecoveryMode` modifier.

Recovery mode can be enabled independently on each chain.

2.2.2 Registries

The system relies on a set of on-chain registries to store cross-chain mappings, module addresses, and price feeds. Each registry provides view functions and `restricted` setters:

- CoreRegistry: (extended by both HubCoreRegistry and SpokeCoreRegistry) Stores the addresses of core system modules (factory, oracle registry, token registry, swap module, flashloan module, caliber beacon, and bridge adapter beacons).
- OracleRegistry: Registers and validates on-chain price feeds (Chainlink). Provides a `getPrice` function that enforces staleness checks to ensure up-to-date token valuations.
- ChainRegistry: Maintains the mapping between EVM chain IDs and Wormhole chain IDs for all supported networks.
- TokenRegistry: Maps local ERC-20 tokens to their cross-chain counterparts and vice versa, enabling token lookups for bridging.

2.2.3 Machine

The Machine is the main entry point. It implements a subset of the ERC-4626 standard, inherits from MakinaGovernable, and is responsible for deposits, withdrawals, share price, fees, and interactions with the Caliber contracts deployed on the Hub chain and Spoke chains.

2.2.3.1 Tokens

The Machine uses two main tokens: an accounting token and a share token.

The accounting token serves as the primary unit for asset valuation and accounting within the system. It is also the token with which deposits are made.

The share token represents user ownership proportionally.

2.2.3.2 Deposits and Withdrawals

Deposits and withdrawals are permissioned and managed through the `deposit` and `redeem` functions.

The deposit process is identical to that of an ERC-4626 but permissioned. Only the designated depositor can call the `deposit` function, which transfers assets from the depositor to the contract and mints shares to the specified receiver based on the current share price. A maximum share limit can be set, after which no more deposits can be made. For slippage protection, the depositor can specify a minimum amount of shares they want to receive.

Withdrawals follow a similar pattern, where only the designated redeemer can call the `redeem` function. The process burns shares from the caller and transfers the corresponding assets to the receiver.

Because of the cross-chain nature of the system, the AUM is always entirely or partially out-of-date, meaning that the share price is not always up-to-date. Furthermore, price updates can be frontrun. The depositor and redeemer are expected to be aware of this and take it into account when allowing deposits and withdrawals. One way they can do this is by implementing a long enough queue time for deposits and withdrawals.

2.2.3.3 Interactions with Calibers

The Machine can send tokens to Caliber contracts through two different mechanisms depending on the target chain. For the Hub chain, it uses the `transferToHubCaliber` function to send tokens directly to the Hub Caliber. For Spoke chains, it uses the `transferToSpokeCaliber` and `sendOutBridgeTransfer` functions to bridge tokens through bridge adapters.

Tokens can flow back to the Machine from Caliber contracts using the `manageTransfer` function. This function can only be called by the Hub Caliber or the bridge adapters. Further details on bridging can be found in the Bridge section.

2.2.3.4 AUM and Share Price Management

The Machine tracks the total Assets Under Management (AUM) through several mechanisms and maintains a cached `_lastTotalAum` value that gets updated during AUM calculations. The contract provides conversion functions where `convertToShares()` converts assets to shares based on current supply and the last total AUM, while `convertToAssets()` converts shares back to assets using the same parameters.

The share price calculation includes a virtual share to protect against share inflation attacks and uses the following formula:

```
shares = assets * (totalSupply + offset) / (lastTotalAUM + 1)
assets = shares * (lastTotalAUM + 1) / (totalSupply + offset)
```

AUM updates happen through the `updateTotalAum` function, which aggregates data from multiple sources:

- Idle token balances held directly in the Machine contract
- Net AUM reported by the Hub Caliber
- Net AUM reported by all registered Spoke Calibers
- Tokens sent by the Machine but not yet received by a Spoke Caliber

- Tokens sent by a Spoke Caliber but not yet received by the Machine

To obtain the cross-chain AUM components (last three points), the Machine leverages Wormhole's Cross-Chain Query (CCQ).

Unlike a standard on-chain bridge message, CCQ begins entirely off-chain:

1. A CCQ request is constructed and submitted by specifying:
   - The list of chains
   - The target contract address on each chain: each Caliber's Mailbox
   - The function to call: `getSpokeCaliberAccountingData()`
2. Guardians execute, sign, and aggregate the responses to get:
   - A bytes blob containing an array of `PerChainQueryResponse` data
   - An array of `GuardianSignature[]` signatures
3. On-Chain Verification

   The Machine's `updateSpokeCaliberAccountingData` function:
   - Gets the list of valid Guardians from the Wormhole endpoint using `getCurrentGuardianSetIndex()`
   - Calls `decodeAndVerifyQueryResponseCd()` from the Wormhole Library to verify that at least 2/3 of the total Guardian Set provided a signature and that all signatures are valid
   - Decodes the `PerChainQueryResponse[]` (including `netAum`, `positions`, `baseTokens`, timestamps, and bridge in/out arrays)
   - Checks for stale (using the configured `caliberStaleThreshold`) or out-of-order data
   - Updates each `SpokeCaliberData` in storage

The maximum staleness that a position can have while still being used for accounting is the sum of the position staleness threshold and the caliber staleness threshold.

2.2.3.5 Fee Management

The `manageFees` function mints and distributes two types of fees in share tokens, subject to a cooldown and a cap:

- Fixed fee, calculated by the external fee manager using the current share supply:

```
 fixedFee = IFeeManager(_feeManager).calculateFixedFee(
    currentShareSupply,
    elapsedTime
);
```

- Performance fee, calculated by the external fee manager using the share-price growth since the last mint:

```
perfFee = IFeeManager(_feeManager).calculatePerformanceFee(
    currentShareSupply,
    _lastMintedFeesSharePrice,
    adjustedSharePrice,
    elapsedTime
);
```

Afterwards, the Machine approves the FeeManager to spend the fees and calls `IFeeManager.distributeFees(fixedFee, perfFee)`, allowing the fee manager to distribute the fees as it sees fit.

Both fees are only minted if the elapsed time since the last mint exceeds `_feeMintCooldown`, and are capped by `maxFee = _maxFeeAccrualRate * elapsedTime`. After minting, any dust balance left in the contract is burned to preserve precision.

2.2.3.6 Configuration Functions

The Machine provides a set of privileged setters that allow for on-chain configuration and emergency controls:

By Governance (`restricted`):

- Core setters: `setFeeManager()`, `setDepositor()`, `setRedeemer()`
- Fee parameters: `setMaxFeeAccrualRate()`, `setFeeMintCooldown()`
- Spoke chains: `setSpokeCaliber()`, `setSpokeBridgeAdapter()`

By the security council:

- `resetBridgingState()`,``setRecoveryMode()``

By the Risk Manager (`onlyRiskManager`):

- `setShareLimit()`

By the Risk Manager Timelock (`onlyRiskManagerTimelock`):

- `setCaliberStaleThreshold()`
- `setOutTransferEnabled()`
- `setMaxBridgeLossBps()`

2.2.4 Pre-Deposit

The PreDepositVault is a temporary contract designed to handle initial user deposits before the main Machine contract is deployed. It allows early participants to deposit assets and receive shares. Once the Machine is ready, the vault's assets and control of the share token are migrated to it in a one-time operation.

2.2.4.1 Deposits and Redemptions

Users can deposit a `depositToken` into the vault via the `deposit` function. This does not have to be the same as the `accountingToken`. In return, users receive `shareToken` corresponding to their share of the deposited assets.

Deposits can be restricted to a specific list of users via a `whitelistMode`. The total number of shares that can be minted is capped by a `shareLimit`, which can be adjusted by the Risk Manager.

Redemptions are handled by the `redeem` function, allowing users to burn their shares to withdraw the corresponding amount of `depositToken`.

2.2.4.2 Migration to Machine

The transition to the Machine contract is done through a two-step migration process:

1. `setPendingMachine`: The core factory sets the address of the new Machine contract.
2. `migrateToMachine`: On construction, the designated Machine contract calls this function to finalize the migration.

During migration, the PreDepositVault transfers its entire balance of the `depositToken` to the Machine and transfers ownership of the `shareToken` contract. After the migration is complete, the `deposit`, `redeem`, and configuration functions in the vault are permanently disabled by the `notMigrated` modifier.

2.2.5 Caliber

The Caliber is an upgradeable smart contract that provides execution and accounting logic for managing financial positions on each chain. It integrates with an external virtual machine (Weiroll VM) to execute sequences of encoded instructions. It relies on Merkle root validation to enforce that only pre-authorized instructions can be used. The contract maintains a registry of active positions, performs token valuation based on price feeds, and applies risk parameters and cooldowns to govern execution behavior.

2.2.5.1 Position Management and Accounting

The Caliber tracks a set of positions, each identified by a unique ID and classified as either an asset or a debt. Each position has an associated value and a timestamp indicating the last time it was accounted for.

The `managePosition` function is the primary method for updating a position. It pairs two instructions: a management instruction that executes a set of operations, and an accounting instruction that determines the new value of the position after those operations.

When `managePosition()` is called, the contract:

- Computes the current value of the position using the accounting instruction.
- Executes the management instruction.
- Re-runs the accounting instruction to determine the updated value of the position.
- Computes the change and validates the result.

To validate the result of a management operation, the contract evaluates:

- Whether the base token balances changed during execution.
- Whether the position is an asset or a debt.
- Whether the position value increased or decreased.

Based on these factors, a slippage check is performed that ensures the instruction did not create a loss larger than the allowed percentage set by the risk manager. Additionally, the contract may reject changes that would indicate inconsistent directionality (e.g., an asset position decreasing without a corresponding balance increase in the Caliber).

If a position reaches zero value, it is removed. If a position is new, it is registered. Otherwise, the updated value and timestamp are recorded.

There is a cooldown timer per management instruction, that limits how frequently a given instruction can be executed.

2.2.5.2 Instruction Validation and Execution

Instructions are executed through a delegate call to the Weiroll VM. Each instruction includes:

- A position ID.
- Instruction type, debt flag, and affected tokens.
- A sequence of commands (encoded as bytes32 values).
- An associated state array (bytes[]).
- A bitmap for selective hashing of state elements.
- A Merkle proof.

Before execution, the instruction is validated against the current Merkle root. The root may be updated by the Risk Manager via a time lock. This validation ensures that only pre-approved instructions can be run. While the address and function selector are always given by the instruction, some parameters (such as an `amount`) can be made freely choosable by the operator. To do this, the bitmap would be set to zero for the state element that corresponds to the `amount`.

For more details on the instruction format, see the Weiroll documentation.

2.2.5.3 Flashloan Management

The Caliber supports the execution of flashloan operations as part of position management.

Flashloans are executed through an external flashloan module. They must be initiated inside a management instruction containing a call to the `requestFlashloan` function, passing along the provider, token amount, and a flashloan instruction.

Upon receiving the request, the flashloan module performs the flashloan and calls the `manageFlashLoan` function of the Caliber, passing along the flashloan instruction and amount.

Within `manageFlashLoan()`, the Caliber:

- Verifies that the call originated from the expected flashloan module.
- Ensures that it is currently within the context of a position being managed.
- Confirms that the flashloan instruction matches the position context (ID and type) and is of the correct instruction type.
- Checks that the instruction is an allowed instruction in the merkle tree.

Once validated, the flashloan instruction is executed and the borrowed tokens are returned to the flashloan module. Flashloans with fees are only supported for Aave V3. For the other flashloan providers, the fee is enforced to be zero.

2.2.5.4 Base Token Handling and Swaps

The Caliber maintains a set of base tokens used for value accounting and instruction execution. Tokens can be added or removed through the RiskManagerTimelock. The accounting token is a special base token used as the unit of account.

The contract integrates with a swap module to perform token swaps via the `swap` and `harvest` functions. Swaps can take any token as input, but must always output a base token. Swaps involving base tokens as input are subject to cooldowns and minimum output value constraints to limit losses due to slippage or a compromised operator. Swaps from non-base tokens have no slippage check. In this case, the operator is trusted not to incur excessive slippage.

2.2.5.5 Valuation and AUM Calculation

The `getDetailedAum` function calculates the net asset value by summing:

- The value of all asset-type positions.
- The value of token balances held in base tokens.
- Subtracting the value of debt-type positions.

Position values are validated to be fresh based on a configurable staleness threshold. Valuation of token balances is done using on-chain price data sourced from a registered oracle in the OracleRegistry. Any tokens that are not base tokens are considered to have no value. The AUM of a Caliber is always at least zero, it cannot be negative.

2.2.6 Bridge

The Bridge subsystem handles cross-chain token movements between the Hub and Spoke chains. It coordinates scheduling, execution, and settlement of transfers, through a set of abstract and concrete contracts that interact with external bridge protocols.

2.2.6.1 BridgeController and Adapter

The main entry point for bridging operations is the abstract BridgeController contract, extended by the Machine on the Hub chain and the CaliberMailbox on Spoke chains. It manages the lifecycle of bridge transfers, including scheduling, authorizing, and executing transfers across chains.

The BridgeController does not integrate directly with any specific bridge protocol. Instead, it relies on a set of BridgeAdapter contracts that implement the specific bridging logic. New bridge adapters can be added to the controller by calling the `restricted` function `createBridgeAdapter`. They can then be enabled and disabled with `_setOutTransferEnabled`, and the maximum loss for bridging can be limited with `_setMaxBridgeLossBps`.

To initiate a transfer, `_scheduleOutBridgeTransfer` is used. It specifies which bridge to use, the destination, the amount, the input and output tokens, and a minimum output amount. The minimum output amount is bounded by the maximum allowed slippage, which is set by the risk manager via `setMaxBridgeLossBps()`. The function then approves the specified adapter for the token and amount and calls `scheduleOutBridgeTransfer()`. The bridge adapter transfers the tokens to itself, assigns an ID to the transfer, and marks the amount as reserved before emitting the transfer hash.

Once the transfer is scheduled, the transfer hash is emitted. The transfer hash can be used to authorize the transfer on the destination chain via `_authorizeInBridgeTransfer` on the controller. This adds the hash to a mapping for later validation in the adapter. The hash is used to ensure that the bridge cannot transmit unexpected data as part of the bridge message. It is a known caveat in Across Bridge that incorrect data could be transmitted by a relayer (at a cost). The hash validation avoids this, as long as the operator only sets correct hashes (does not act maliciously).

After being approved, the transfer can be sent via the `_sendOutBridgeTransfer` function. This function calls `sendOutBridgeTransfer()` on the relevant bridge adapter, which is only defined in the specific BridgeAdapter (e.g., AcrossV3BridgeAdapter), and is responsible for executing the transfer on the bridge protocol.

To receive funds, the external bridge's relayer calls a public function on the adapter (e.g., `handleV3AcrossMessage`), which in turn calls the internal `_receiveInBridgeTransfer` function. This function verifies that the incoming message's hash is present in the `_expectedInMessages` mapping. If valid, it creates an InBridgeTransfer record, adds a new ID to a `_pendingInTransferIds` set, and holds the received tokens.

The `claimInBridgeTransfer` function is called by the destination controller to finalize the receipt of funds. It finds the corresponding InBridgeTransfer record, removes it from the pending set, and invokes the `manageTransfer` function on its controller (`msg.sender`) to deliver the assets. If a transfer fails, it can be canceled by calling `cancelOutBridgeTransfer()`. This function returns the escrowed funds to the controller by calling `manageTransfer` with a refund flag. Before `claimInBridgeTransfer` can be called, a Wormhole CCQ message that relays the increased `CaliberOut` data must be received by the Machine, otherwise, the transfer will revert, as it will look as if funds are received that were never sent.

2.2.6.2 CaliberMailbox

The CaliberMailbox contract is deployed on each Spoke chain and serves as the local endpoint for all bridging operations, replacing the Machine's role. It inherits from BridgeController, giving it the ability to manage its set of BridgeAdapter contracts for that specific chain.

The `manageTransfer` function has dual functionality based on the identity of the caller (`msg.sender`). When `manageTransfer` is called by the local Caliber contract, it signifies an outbound transfer from the Spoke to the Hub. In this case, the function pulls tokens from the Caliber and uses the inherited

`_scheduleOutBridgeTransfer` logic to begin the bridging process. Conversely, when `manageTransfer` is called by a BridgeAdapter, it signals the arrival of a completed inbound transfer from the Hub, and the function forwards the received assets directly to the Caliber contract.

2.2.6.3 Bridging Flows

Hub to Spoke:

1. Schedule: Initiated by `Machine.transferToSpokeCaliber`, which schedules the transfer via `BridgeAdapter.scheduleOutBridgeTransfer`.
2. Authorize: `CaliberMailbox.authorizeInBridgeTransfer` on the Spoke side, using the emitted transfer hash.
3. Send: Execute the transfer from the Hub by calling `Machine.sendOutBridgeTransfer`.
4. Claim: Finalize receipt on the Spoke via `CaliberMailbox.claimInBridgeTransfer`.

Spoke to Hub:

1. Schedule: Initiated by `CaliberMailbox.manageTransfer`, scheduling via `BridgeAdapter.scheduleOutBridgeTransfer`.
2. Authorize: `Machine.authorizeInBridgeTransfer` on the Hub side with the transfer hash.
3. Send: Execute the transfer from the Spoke by calling `CaliberMailbox.sendOutBridgeTransfer`.
4. Claim: Finalize receipt on the Hub via `Machine.claimInBridgeTransfer`.

2.2.7 VERSION 2

This section describes the changes made in Version 2 of the contracts.

2.2.7.1 MakinaGovernable

An only mechanic modifier was added to the `MakinaGovernable` contract.

2.2.7.2 Machine

The `transferToHubCaliber`, `transferToSpokeCaliber`, and `sendOutBridgeTransfer` function are now gated by the `onlyMechanic` modifier, removing the previously used `msg.sender` check.

2.2.7.3 Caliber

Grouped Positions

Positions now have a groupId. The single-position accounting method (`accountForPosition`) now reverts if invoked on a position belonging to a group containing more than one member, enforcing that linked positions cannot be updated separately. All grouped position updates must go through the refactored `accountForPositionBatch` function, which ensures that all open positions in the group are updated. Finally, when a position is managed using `managePosition` or `managePositionBatch`, the other positions in the group are invalidated, forcing them to be updated in a batch afterwards.

Pull-Style Transfers & Reentrancy Guard

Funds are now transferred between the MachineEndpoint and Caliber contracts using a pull model. The MachineEnpoint gives an allowance to the Caliber and calls the new `notifyIncomingTransfer` function which moves the tokens into the Caliber. `notifyIncomingTransfer` is protected by a reentrancy guard.

Interface Changes

The `accountForPositionBatch` and `managePositionBatch` functions now returns a list of `values` and `changes` in line with the `accountForPosition` and `managePosition` functions, in addition to taking the additional new `groupIds` parameter.

2.2.8 VERSION 4

This section describes the changes made in Version 4 of the contracts.

2.2.8.1 Machine Fees

The single `maxFeeAccrualRate` was replaced by two independent per-second percentage caps, which `manageFees` uses to cap each fee component separately:

- `maxFixedFeeAccrualRate()`
- `maxPerfFeeAccrualRate()`

The following setters are now `onlyRiskManagerTimelock`:

- `setMaxFixedFeeAccrualRate(uint256)`
- `setMaxPerfFeeAccrualRate(uint256)`
- `setFeeMintCooldown(uint256)`

2.2.8.2 MachineShare

`burn()` now allows self-burns by the holder. Burning from another account is still only possible by the contract owner (expected to be another contract).

2.2.8.3 Stricter loss checks & rounding

In `_receiveInBridgeTransfer`, the BridgeAdapter now enforces a maximum loss bound based on the `inputAmount` specified in the `message`. Concretely, this means that at receipt, the adapter now fetches the per-bridge maximum loss via `IBridgeController.getMaxBridgeLossBps` and requires `receivedAmount >= ceil(inputAmount * (MAX_BPS - maxBridgeLossBps) / MAX_BPS)` reverting with `MaxValueLossExceeded` if it fails. This change addresses a message-integrity gap in Across reported by Sigma Prime. A malicious operator could authorize a hash where `minOutputAmount` was weakened (e.g., set to zero, while `inputAmount` was kept to a high value), and deliver it with only dust on the destination chain. The adapter would accept such a transfer, and the accounting would propagate to the hub. The total AUM would be decreased by `spokeCaliberData.caliberBridgesIn` (`inputAmount` on the destination chain) and increased by `spokeCaliberData.netAum` (the actually received dust). Note that this check compares the amount of `outputToken` and `inputToken`, which requires that the `inputToken` and `outputToken` have the same value and same decimals in both chains.

Additionally, minimum output amounts are now rounded up when calculating the maximum allowed losses. This ensures that losses can never exceed the configured maximum due to rounding. This change was made in response to a finding by Enigma Dark.

2.2.8.4 Factories & Deterministic Deployment (CREATE3)

Changes were introduced to deploy cross-chain contracts (`Caliber`, `CaliberMailbox` and bridge adapters) deterministically at the same address across the different chains using CREATE3. For this, a new `Create3Factory` base contract was introduced, and is used by both `CaliberFactory` and `BridgeAdapterFactory`. The factories now deploy beacon proxies via CREATE3, making the resulting addresses depend only on the factory address and a caller-provided salt.

2.2.8.5 Access control changes

The `resetBridgingState` function is now called by the security council. Previously, it was called by governance (`restricted`).

2.3 Trust Model

2.3.1 Tokens

Tokens that are whitelisted in the system, including base tokens, accounting tokens, and deposit tokens, must be standard ERC20 tokens without special functionality, such as rebasing, transfer hooks (reentrancy), or fees-on-transfer. Tokens must have at most 18 decimals.

2.3.2 Roles and Privileges

2.3.2.1 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: The Governance role has the power to steal all user funds by replacing critical contracts like the Machine or Calibers with malicious versions.

2.3.2.2 Mechanic

Trust Level: Partially trusted.

Capabilities: The Mechanic acts as the Operator role, except during recovery mode. It is responsible for executing routine tasks, such as managing positions in the Caliber contracts and executing scheduled bridge transfers via `sendOutBridgeTransfer`. The system is designed to avoid a total loss of funds, even if the mechanic's private keys become compromised. A compromised or malicious Mechanic cannot directly steal funds, as they do not control withdrawal addresses of positions or critical system parameters. However, they could cause losses by executing approved instructions, incurring at most the maximum loss configured by the risk manager. Monitoring is assumed to be in place to quickly detect and replace a compromised Mechanic.

2.3.2.3 Security Council

Trust Level: Fully trusted.

Capabilities: The Security Council can veto instruction root updates. It can also halt most normal operations by enabling recovery mode. In this mode, it assumes Operator privileges from the Mechanic. It cannot directly withdraw funds. It could cause a denial of service, disallowing withdrawals until the Governance steps in. Additionally, it has the power to reset the bridging state. In the worst case, this could lead to incorrect AUM calculations, which could cause an incorrect machine share price.

2.3.2.4 Redeemer

Trust Level: Partially trusted.

Capabilities: The Redeemer handles withdrawals for users. In the worst case, a malicious Redeemer could steal the funds a user wants to redeem, as they can specify the `receiver` of the redemption when burning the user's shares. This role is expected to be held by a smart contract.

2.3.2.5 Depositor

Trust Level: Partially trusted.

Capabilities: The Depositor handles deposits for users. In the worst case, a compromised Depositor could steal the funds a user wants to deposit, instead of depositing it to the Machine as expected. This role is expected to be held by a smart contract.

2.3.2.6 Fee Manager

Trust Level: Partially trusted.

Capabilities: The Fee Manager determines the fees and distributes them. In the worst case, a compromised Fee Manager could mint the maximum configured fee and distribute it to itself. This role is expected to be held by a smart contract.

2.3.2.7 Risk Manager & Risk Manager Timelock

Trust Level: Partially trusted.

Capabilities: These roles can alter specific system parameters. The Risk Manager can instantly change the `shareLimit`, while the Risk Manager Timelock can change critical parameters like `caliberStaleThreshold` and `maxBridgeLossBps` after a delay. A malicious actor in these roles cannot directly steal funds but could disrupt operations, for example by setting the `shareLimit` to zero to block deposits or by setting `maxBridgeLossBps` to an unfeasible value to halt all bridging activity. If it adds a malicious instruction to the merkle tree, this can be vetoed by the security council or the instruction root guardians.

2.3.2.8 Instruction Root Guardians

Trust Level: Partially trusted.

The Instruction Root Guardians can veto an instruction root update that has been queued by the risk manager. In the worst case, they can temporarily stop new instructions from being added, until Governance steps in. Additionally, they could not veto a bad instruction, which would be catastrophic, but only if all guardians and the security council did not veto it. As long as there is one veto, an instruction can be denied.

2.3.2.9 End Users

Unpermissioned individuals who hold the system's share tokens.

Trust Level: Untrusted.

Capabilities: End users have no execution rights within the core contracts. They can only hold and transfer their share tokens. They cannot deposit or redeem assets directly from the Machine.

2.3.3 Upgradeability

The core contracts, including Machine, Caliber, and CaliberMailbox, are upgradeable. The entity that controls the proxy's upgrade mechanism (typically the Governance role) can change the entire logic of the system at any time. In the worst case, all deposited funds could be stolen.

2.3.4 External Dependencies

The system's security also depends on the correctness and integrity of several external components:

- Oracles: The system relies on Chainlink price feeds registered in the OracleRegistry for all asset valuation and AUM calculations. These oracles are assumed to provide accurate and timely price data. A manipulated or lagging oracle feed could lead to significant miscalculations of the share price, enabling theft or causing user losses.

- External Bridge Protocols: The BridgeAdapter contracts integrate with third-party bridges (e.g., Across). The system trusts that these external protocols are secure. A vulnerability in an underlying bridge could lead to the permanent loss of all funds being transferred through it.
- Wormhole: AUM data from Spoke chains is relayed to the Machine via messages signed by Wormhole guardians. The system trusts that at least 2/3 of the Wormhole guardian set is secure and will only sign valid, untampered messages.
- L2 Infrastructure: For all Spoke chains, the system implicitly trusts the integrity of that network's sequencer and data availability layer. A malicious or faulty sequencer could censor or reorder transactions, impacting the overall system's AUM or causing a denial of service.

2.3.5 Instructions and Integrations with External Protocols

When integrating the Caliber with external systems it is critical to ensure that every new instruction committed to the Merkle root has been thoroughly audited. No specific instructions were in scope for this audit. Similarly, the accounting instruction that computes position values (and thus contributes to AUM) must be designed to withstand adversarial or unexpected states in those external systems. All external systems that hold funds are fully trusted and can in the worst case steal all funds that have been deposited to them. For external contracts that can be upgraded, the role that can execute the upgrade is fully trusted.

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	0
Low-Severity Findings	2

5.1 Base Token Removal Can Be Blocked

Design Low Version 1 Risk Accepted

CS-MACO-007

The `removeBaseToken()` in Caliber checks that the balance of the contract is zero before allowing the removal of the base token. This can be problematic, as the removal can be blocked by anyone sending 1 wei of tokens to the contract. The access control is `onlyRiskManagerTimeLock`, which means it is intended to be called by a timelocked contract. While the call is pending execution, there is time for anyone to send tokens.

Blocking removal of a base token could be a problem if the oracle for the token is no longer available. This would block accounting, as well as block swaps of the token.

Risk accepted:

Makina is aware of the issue and has decided not to change the behavior, accepting the risk of token removal being blocked.

5.2 End-of-state Sentinel Comparison Can Incorrectly Decode

Design Low Version 1 Acknowledged

CS-MACO-009

The `_decodeAccountingOutputState` function uses a fixed 32-byte sentinel to detect the end of the output state:

```
if (bytes32(state[i]) == ACCOUNTING_OUTPUT_STATE_END) {
    break;
}
```

where the sentinel is defined as:

```
bytes32 private constant ACCOUNTING_OUTPUT_STATE_END = bytes32(type(uint256).max);
```

This check can trigger a premature break in two scenarios:

1. Truncation of dynamic `bytes`: Casting a `bytes` element to `bytes32` truncates any input longer than 32 bytes (and right-pads shorter inputs with zeros). If the first 32 bytes of a longer element happen to equal `0xff…ff`, it will incorrectly match the sentinel and terminate the loop early.
2. Collision with valid `–1` outputs: The sentinel value `bytes32(type(uint256).max)` is the two's-complement representation of `–1` (`0xff…ff`). Any legitimate output intentionally encoding –1 will be mistaken for the end marker, causing valid elements to be skipped.

Acknowledged:

Makina has acknowledged the limitations of the sentinel check, but has decided they are not problematic, with the following response:

```
The finding describes two cases that were not deemed problematic, based on the assumption that the
output state of authorized accounting instructions must contain only priceable token amounts in relevant
slots, represented as unsigned integers, which in Solidity are limited to a maximum size of 32 bytes.
```

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	2

- Bridge Transfer Can Be Counted as Profit Code Corrected
- Reentrancy Can Cause Incorrect Share Price Code Corrected

Medium-Severity Findings	5

- Invalidation of All Ungrouped Positions When Managing an Ungrouped Position Code Corrected
- Hash Collision in _getStateHash Code Corrected
- Linked Positions Can Be Updated Separately Code Corrected
- Reentrancy Can Circumvent Slippage Check Code Corrected
- onFlashLoan Reverts Due to Missing Approval Code Corrected

Low-Severity Findings	5

- Caliber Data Can Be Overwritten With Older Value Code Corrected
- Inconsistent Staleness Check Code Corrected
- Incorrect Natspec Specification Changed
- Incorrect Storage Location Code Corrected
- outBridgeTransferCancelDefault Is Underspecified Specification Changed

Informational Findings	5

- ExpectedDataHash Could Be Cleared Earlier Code Corrected
- Incompatible Modifier Usage Code Corrected
- Redundant Loop Counter Code Corrected
- Registry Cannot Be Correctly Updated Code Corrected
- Unchecked Loops Code Corrected

6.1 Bridge Transfer Can Be Counted as Profit

Correctness High Version 1 Code Corrected

CS-MACO-001

In CaliberMailbox, the `manageTransfer` function transfers tokens from the BridgeAdapter to the Caliber. This inflow of tokens can be counted as profit in the Caliber's slippage check if it is triggered through a reentrant call. As a result, a malicious operator can cause a large loss, which is offset by the transfer from the Bridge to pass the slippage check.

Consider the following example attack:

1. The operator calls `Caliber.swap()`, using a base token as input. The operator chooses a swap route that contains a malicious pool with high slippage (up to 100%). Additionally, the swap route calls the `transfer` function of a malicious token controlled by the operator.
2. The `transfer` function calls back into the operator's contract, which calls `CaliberMailbox.manageTransfer()`. This transfers tokens into the Caliber (which were previously bridged from the Machine). The tokens transferred are the same as the output token used in the swap.
3. The slippage check in `Caliber.swap()` is triggered, checking the contract's balance of output tokens. The transferred tokens are counted, making it look as if there was no slippage, even though the swap route had high slippage.
4. The slippage incurred is a profit for the malicious operator. This amount is not bounded by the slippage check, as would be intended.

In summary, a malicious/compromised operator can bypass the slippage check by reentering into the CaliberMailbox. Although both functions are protected by a reentrancy guard, they are in separate contracts, allowing reentrancy from one to the other. At most, 50% of the AUM could be stolen in one transaction, if 50% is lost through slippage, and 50% is transferred in through the bridge.

There are more possible variations of the same attack: Instead of `swap()`, `managePosition()` with an instruction that allows reentrancy could be used. Instead of `manageTransfer()`, `resetBridgingState()` could be used, (which is only callable by the `restricted` role, not the operator) as it also transfers funds to the Caliber. The `transferToHubCaliber` function could also be used, as it transfers funds from the Machine to the Caliber.

Code corrected:

The following changes were made to the code to prevent this issue:

- A `notifyIncomingTransfer()` non-reentrant function was added in Caliber to handle reception of incoming funds. This ensures that the accounting cannot be done in a reentrant call originating in the Caliber.
- Machine or CaliberMailbox now approve the Caliber and call `notifyIncomingTransfer()` instead of directly transferring funds to it.
- Checking that the transferred token is a registered base token in the Caliber is now performed in `notifyIncomingTransfer()`.

6.2 Reentrancy Can Cause Incorrect Share Price

Correctness | High | Version 1 | Code Corrected

CS-MACO-002

The share price of the Machine depends on the total AUM, which is queried by calling `getDetailedAUM()` on the Caliber contracts. For the Hub Caliber, this call is made directly on-chain, whereas for the Calibers on other chains, it is read through Wormhole. As the accounting can be triggered at any time, it can also be called during a reentrant call. This can lead to a situation where there is an intermediate state that is reflected in the AUM of the Hub Caliber, leading to a share price that is too high or too low. Depending on the setup for depositing and withdrawing from the Machine, an incorrect share price can lead to a loss of funds, or to incorrect performance fees for the operator.

Consider the following attack scenario, which can reduce the share price:

1. The operator initiates a swap in the Hub Caliber, transferring the contract's total balance of a base token to the SwapModule.

2. One of the tokens on the swap path is a malicious token, with a `transfer` function that calls into `Machine.updateTotalAUM()`. This calls `getDetailedAUM()` on the Hub Caliber.
3. `getDetailedAUM()` will read the base token balances of the Caliber. As the tokens are currently in the SwapModule, the balance will be `0`.
4. The AUM will be lower than it should be, leading to a lower share price until the accounting is updated again.

The operator can trigger this attack since they typically control the swap path. A third party could also execute it by deceiving the operator into routing the swap through their token. This may be possible if the path through this token gives the best rate, which an attacker could cause by providing liquidity for their malicious token. Swap aggregators may route through unknown tokens to optimise for the best swap rate.

Note that both the `Machine.updateTotalAUM()`, as well as the `Caliber.swap()` function, are marked as `nonReentrant`, but this only prevents reentrancy from the same contract. It does not prevent reentrancy from other contracts, which is the case here. Additionally, the `getDetailedAUM()` function is a view function, meaning it does not write any state. However, the returned value is written to the state of the Machine. This is an example of a so-called read-only reentrancy.

Depending on the allowed flashloan instructions, it may also be possible to increase the share price. This could be done by taking a flashloan of a base token, then doing some call to untrusted code, which reenters into `Machine.updateTotalAUM()` while the base tokens are still in the Caliber. This could arbitrarily increase the AUM of the Machine, as it would count the flashloaned token balance. This variation is only possible if there is an allowed instruction that allows a reentrant call, but does not move the flashloaned tokens out of the Caliber. An alternative to counting the tokens in the balance would be to call `Caliber.accountForPosition` from within a flashloan, while the flashloaned tokens are temporarily in the position. A subsequent call to `updateTotalAum()` would count the position with the flashloaned amount in the AUM.

In summary, reentrancy into the accounting logic can lead to incorrect AUM calculations.

Code corrected:

A read-only reentrancy guard was added to `Caliber.getDetailedAum()`, which ensures that it cannot be called from within a management instruction. This solves the issue.

6.3 Invalidation of All Ungrouped Positions When Managing an Ungrouped Position

Correctness | Medium | Version 2 | Code Corrected

CS-MACO-023

In the Version 2 of the Caliber implementation, positions are added to `_positionIdGroups[groupId]` in the `_accountForPosition` function as follows:

```
} else if (currentValue > 0) {
    pos.value = currentValue;
    pos.lastAccountingTime = block.timestamp;
    if (lastValue == 0) {
        pos.isDebt = instruction.isDebt;
        $._positionIds.add(posId);
        $._positionIdGroups[groupId].add(posId);
        emit PositionCreated(posId, currentValue);
```

```
    } else {
        emit PositionUpdated(posId, currentValue);
    }
}
```

Afterwards, all members of a group are invalidated in `_managePosition` by the call to the `_invalidateGroupedPositions` function but neither path guards against `groupId == 0`. As a result, every position that is not in a group ends up in the zero group, and managing any ungrouped position deletes the `lastAccountingTime` for every other ungrouped position.

Code corrected:

The code has been corrected to ensure that positions are not added to the zero group and that `_invalidateGroupedPositions` is not called for the zero group.

6.4 Hash Collision in _getStateHash

Correctness **Medium** **Version 1** **Code Corrected**

CS-MACO-003

The function `_getStateHash` builds a hash over selected `state` elements by concatenating their raw `bytes` values and then applying `keccak256`. However, because no delimiters or length prefixes are included between concatenated elements, different sequences of inputs can produce an identical concatenated blob and thus yield the same hash.

Consider two different `state` arrays and a bitmap that selects all elements in each:

1. Allowed `state`

```
state = [
  hex"a",
  hex"bc"
];
```

   The concatenation yields: `hashInput = "a" || "bc"  = "abc"`

2. Different `state` :

```
state = [
  hex"ab",
  hex"c"
];
```

   The concatenation yields: `hashInput = "ab" || "c" = "abc"`

Since both hash inputs are byte-for-byte identical (`"abc"`), their hashes collide despite originating from different arrays, allowing an operator to manipulate state that should be fixed.

Code corrected:

The code has been updated to now hash each element separately before concatenating them, as follows:

```
hashInput = bytes.concat(hashInput, keccak256(abi.encodePacked(state[i])));
```

This solves the hash collision issue by ensuring that each element contributes uniquely to the final hash, preventing different sequences of inputs from producing the same hash.

In Version 3, the redundant `abi.encodePacked` was removed:

```
hashInput = bytes.concat(hashInput, keccak256(state[i]));
```

6.5 Linked Positions Can Be Updated Separately

Design Medium Version 1 Code Corrected

CS-MACO-004

As of Version 1, an accounting instruction always updates a position independently of other positions. However, some positions are "linked", in the sense that they must always be updated together. For example, a collateral position and its corresponding debt position are linked, as a liquidation of the collateral will also reduce the debt of the position.

Consider the following example:

- A position is liquidated, which reduces the collateral and debt of the position.
- The debt position is accounted to reflect the liquidation, but the collateral position is not accounted at the same time.
- If the AUM is queried now, it will temporarily increase by the liquidation amount, as the debt is reduced but the loss of collateral is not yet accounted for.

The opposite case could also happen, where the AUM would temporarily decrease if the collateral is accounted for, but the debt is not.

Code Corrected:

A new `groupId` field was added to the `Instruction` struct an a new mapping `mapping(uint256 ⇒ EnumerableSet.UintSet) _positionIdGroups` in the Caliber's storage, enabling the contract to record exactly which open positions belong to each group. The single-position accounting method (`accountForPosition`) now reverts with `PositionIsGrouped` if invoked on a position belonging to a group containing more than one member, enforcing that linked positions cannot be updated separately.

All grouped position updates must go through the refactored `accountForPositionBatch` function, which ensures that all open positions in the group are updated at the same time.

Finally, when a position is managed using `managePosition` or `managePositionBatch`, the other positions in the group are invalidated, forcing them to be updated in a batch afterwards.

However, in some scenarios, positions can become unlinked. See Positions can become unlinked for more details.

6.6 Reentrancy Can Circumvent Slippage Check

Correctness Medium Version 1 Code Corrected

CS-MACO-005

In Caliber, the slippage check is done by accounting a position before and after a management instruction is called, then comparing the difference. However, if there is a reentrant call to `accountForPosition()` (which does not have a reentrancy guard), the difference can be incorrect.

Consider the following example attack scenario, allowing an operator to cause unchecked losses:

1. The operator calls `managePosition()`, which calls `accountForPosition()` to log the value of the position.
2. The management instruction adjusts the position, then calls untrusted external code (e.g. a token in a swap). This code can reenter into `accountForPosition()`, which will log the value of the position again.
3. After the management instruction is finished, `accountForPosition()` is called again, comparing to the latest log. If the second log happened after the position was adjusted, the difference will be zero, even though the position was adjusted.
4. As the `change` variable returned by `accountForPosition()` will be zero. If `change` is zero, `_checkPositionMaxDelta()` will never revert, allowing any amount of slippage to occur, as long as the `affectedTokensValue` has not decreased. This means there could be a large amount of debt added without receiving any tokens in return, while not triggering the slippage check.

Depending on when the reentrant call happens, the difference can be zero, or larger/smaller than it should be, which could circumvent different slippage scenarios. The reentrant calls possible depend on the allowed instructions. However, it should be assumed that calls to external code can generally reenter, unless it has specifically been vetted not to do so. Swaps are the most common source of reentrancy, as they can occur anywhere on a swap route.

Code corrected:

A reentrancy guard has been added to `accountForPosition()` and `accountForPositionBatch()`. This ensures that the function cannot be called during a call to `managePosition()`, which resolves the issue.

6.7 onFlashLoan Reverts Due to Missing Approval

Correctness | Medium | Version 1 | Code Corrected

CS-MACO-006

In the FlashLoanAggregator contract, the `onFlashLoan` function should approve the DSS Flash contract, so that it can pull the funds to repay the flashloan. However, it incorrectly transfers the funds instead of approving them. This will make the flashloan revert, rendering the DSS Flash integration unusable.

Code corrected:

Flashloan repayment in `onFlashLoan()` was fixed by approving funds to sender instead of transferring them.

6.8 Caliber Data Can Be Overwritten With Older Value

Design | Low | Version 1 | Code Corrected

CS-MACO-008

In MachineUtils, the `_handlePerChainQueryResponse` function takes a wormhole CCQ response and writes to the `caliberData`. The timestamp of the response is checked, but a timestamp that is equal to the timestamp of the existing data is allowed. As a result, if there are two responses with the same timestamp, the newer response can be overwritten by an older one.

On some blockchains, such as Arbitrum, the block time is less than a second. This means that two subsequent blocks can have the same timestamp. In this case, data that is one block older can overwrite the data of a newer block.

Code corrected:

The check for the timestamp in `_handlePerChainQueryResponse` was changed to only allow strictly newer timestamps, preventing overwriting of newer data with older data.

6.9 Inconsistent Staleness Check

Correctness **Low** **Version 1** **Code Corrected**

CS-MACO-010

The position accounting staleness check in Caliber is not always done the same way.

In `getDetailedAUM()`, the function reverts if `lastAccountingTime <= _positionStaleThreshold`.

However, in `isAccountingFresh()`, the function returns false if `lastAccountingTime < _positionStaleThreshold`.

If the time is equal, the two functions behave differently.

Code corrected:

Position accounting in `Caliber.isAccountingFresh()` was made stricter, using `>=` instead of `>`, which aligns with the similar check in `Caliber.getDetailedAum()`.

6.10 Incorrect Natspec

Correctness **Low** **Version 1** **Specification Changed**

CS-MACO-011

In IMachine, `setMaxFeeAccrualRate()` is documented as setting a fee accrual rate in basis points. However, the maximum rate is actually a number of shares per second, not a percentage.

Specification changed:

The natspec for `setMaxFeeAccrualRate()` has been updated to correctly reflect it behavior:

```
//@param newMaxFeeAccrualRate The new maximum fee accrual rate in wei per second.
```

6.11 Incorrect Storage Location

Correctness | Low | Version 1 | Code Corrected

CS-MACO-012

In OracleRegistry, there is the following comment and code:

```
// keccak256(abi.encode(uint256(keccak256("makina.storage.OracleRegistry")) - 1)) & ~bytes32(uint256(0xff))
bytes32 private constant OracleRegistryStorageLocation =
    0x1fbdc0014f4c06b2b0ff2477b8b323f2857bce3cafc75fb45bc5110cee080300;
```

The comment correctly states the intended storage location, but the hex value is incorrect. The hex value was accidentally copied from the storage location of the ChainRegistry contract.

Code corrected:

The value for `OracleRegistryStorageLocation` was corrected to the intended value, as per the comment.

6.12 outBridgeTransferCancelDefault Is Underspecified

Design | Low | Version 1 | Specification Changed

CS-MACO-013

The Natspec for `outBridgeTransferCancelDefault()` states the following:

```
/// @notice Returns the default amount that must be transferred to the adapter to cancel an outgoing bridge transfer.
/// @dev If the transfer has not yet been sent or if the full amount was refunded by the external bridge, returns 0.
/// @dev If the bridge retains a fee upon cancellation, the returned value reflects that fee.
```

However, there is a case that is not explicitly mentioned: A bridge transfer could have been sent to the bridge, but not (yet) filled. In this case, the bridge may refund the funds later. In this situation, the implementation in AcrossV3BridgeAdapter will return the full amount of the transfer, which implies a 100% fee. When the bridge returns the funds later, the return value will correctly become much smaller.

The behavior of the return value may be confusing. There could be a situation where it is already known that the transfer has passed its deadline and will be refunded, but the refund has not yet happened yet. Refunds in Across are batched, so there can be a delay. If the operator calls `outBridgeTransferCancelDefault()` in this situation, it will return the full amount of the transfer, which may be interpreted as a fee based on the Natspec.

The Natspec should clearly specify what the return value means in the case of pending refunds.

Specification changed:

The specification has been updated to clearly specify the behavior, as follows:

```
/// @dev If the bridge retains a fee upon cancellation and only a partial refund was received, the returned value reflects that fee.
/// @dev In all other cases (e.g. including pending refunds or successful bridge transfers), returns the full amount of the transfer.
```

6.13 ExpectedDataHash Could Be Cleared Earlier

Informational | Version 1 | Code Corrected

CS-MACO-015

In FlashloanAggregator, the `ExpectedDataHash` is cleared after the flashloan returns. However, it could be cleared earlier, right after the hash is checked in the flashloan callback. This would reduce the risk of reentrancy. Reentrancy is already only possible with exactly the same data, which we have not identified any problems with, but it is still a good practice to clear the hash as early as possible.

Code corrected:

`ExpectedDataHash` is now always cleared directly after being verified by `_isValidExpectedDataHash`.

6.14 Incompatible Modifier Usage

Informational | Version 1 | Code Corrected

CS-MACO-016

In CaliberMailbox, the `authorizeInBridgeTransfer` function uses the modifiers `notRecoveryMode` and `onlyOperator`. When recovery mode is not active, the operator is always the mechanic. As a result, the `onlyOperator` modifier should use `onlyMechanic` instead, as the function cannot be used when the operator is not the mechanic.

Code corrected:

The following changes have been made to clarify the access control:

- An `onlyMechanic` modifier was added to MakinaGovernable and used in CaliberMailbox's `authorizeInBridgeTransfer()` for clarity.
- `onlyMechanic` is now also used in Machine's `transferToHubCaliber()`, `transferToSpokeCaliber()` and `sendOutBridgeTransfer()`.

6.15 Redundant Loop Counter

Informational | Version 1 | Code Corrected

CS-MACO-019

The `_decodeAccountingOutputState` of the Caliber contract function maintains two counters, `i` and `count`, which are always incremented together, rendering one of them redundant.

Code corrected:

The code has be refactored to use only one counter, `i`.

6.16 Registry Cannot Be Correctly Updated

Informational | Version 1 | Code Corrected

CS-MACO-020

In ChainRegistry, `setChainIds()` does not allow a zero value to be set. It sets two values, which should refer to each other. In case one of the values is updated later to point to a new third value, it will be impossible to reset the other one to zero, leaving an incorrect pointer.

Consider this example where `A` is updated:

1. `A -> B, B -> A`
2. `A -> C, C -> A, B -> A`.

In the example, `B` should now point to id zero, but this is not allowed to be set.

The same issue is also present in `TokenRegistry.setToken()`.

Code corrected:

Deprecated entries are now cleaned up in `setToken()` and `setChainIds()`.

6.17 Unchecked Loops

Informational **Version 1** **Code Corrected**

CS-MACO-021

In multiple places, the code uses unchecked arithmetic increment patterns in for loops as follows:

```
for (uint i = 0; i < array.length;) {
    acc += array[i];
    unchecked { i++; } // i gets incremented without overflow checks -- less gas used
}
```

Solidity 0.8.22 introduced an overflow check optimization that automatically generates an unchecked arithmetic increment of the counter of for loops. This new optimization removes the need for unchecked increment patterns in for loop bodies. See Solidity 0.8.22 Release Announcement.

Code corrected:

The code has been refactored to remove the unchecked increment pattern in for loops.

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 Bad Debt Can Underestimate AUM

Informational **Version 1**

CS-MACO-014

In the Caliber, the AUM is calculated by summing up the balances of all positions and tokens in the Caliber. If there is a pair of collateral and debt positions that is in a state of "bad debt", meaning the collateral is worth less than the debt, the sum of the two positions will be negative. However, the sum should actually be zero, as the debt does not need to be repaid. The total AUM of the Caliber is enforced to be at least zero, but if there are other positions that have a positive value, the AUM will be underestimated by reducing them by the bad debt.

7.2 Operator Can Cause Invalid Bridge State

Informational **Version 1**

CS-MACO-017

The operator can use the `cancelOutBridgeTransfer` function to cancel a bridge transfer. In order to call it, there must be excess tokens in the BridgeAdapter. The operator can initiate and process a 1 wei bridge transfer, donate 1 wei of tokens, then incorrectly cancel this transfer. This will lead to `MachineOut < CaliberIn`, which will block the accounting of the Machine share price and lead to a stale share price.

The operator can also achieve a similar accounting mismatch by submitting an incorrect message hash in `authorizeInBridgeTransfer()` and then processing an unexpected bridge message.

The operator cannot directly profit from this and can be removed by governance, so there is no incentive to do this. The bridge state can be repaired by governance calling the `resetBridgingState` function.

7.3 Overflow in previewRedeem()

Informational **Version 1**

CS-MACO-018

In the `maxDeposit()` function of the PreDepositVault contract, the function `previewRedeem` is called with `_shareLimit` to compute the maximum asset limit:

```
uint256 _assetLimit = previewRedeem($._shareLimit);
```

Where previewRedeem performs the following calculation:

```
return shares.mulDiv((dtBal * price_d_a) + dtUnit, price_d_a * (stSupply + 10 ** $._shareTokenDecimalsOffset));
```

Setting a very large `_shareLimit` could cause an overflow, depending on token balances and price value. The vault currently avoids this when *_shareLimit == type(uint256).max* by directly returning *type(uint256).max*, but for large non-max values this overflow remains theoretically possible.

7.4 Users May Exit Before Fees Are Minted

Informational **Version 1**

CS-MACO-022

The system accrues fees over time but applies them only when the `manageFees` function is called after a cooldown period (`_feeMintCooldown`). At that point, new shares are minted and distributed to the fee recipients. This dilutes existing holders proportionally.

Because fees are applied discretely, users who redeem before fees are minted avoid paying their share of the pending fees. Conversely, users who deposit before fees are minted will be charged the full pending fee, even though they were only deposited for a fraction of the time since fees were last charged. The profitability of intentionally avoiding fees depends on the value `_feeMintCooldown` and the implementation of the `depositor` and `redeemer`.

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 Chainlink Oracle Considerations

Note Version 1

The OracleRegistry uses Chainlink oracles to price assets. The following should be taken into consideration when configuring feed routes:

- When using a feed route that utilizes two price feeds chained together, the maximum deviation without a price update triggering can be the maximum deviation of both feeds combined. Any oracle deviation from the current market price can lead to a looser slippage check and inaccuracy in AUM calculations.
- The `feedStaleThreshold` should be set to at least the heartbeat interval of the oracle. This avoids interpreting a price as stale when the oracle is updating at its expected maximum interval.

8.2 Collusion of Trusted Roles Can Escalate Privileges

Note Version 1

Collusion between the three roles risk manager, security council, and root instruction guardians, could escalate their privileges. In particular, the Risk Manager role could add a malicious `ACCOUNTING` type instruction that allows draining funds if the security council and instruction root guardians do not veto it. As accounting is permissionless, this instruction could then be executed even without the operator.

Each of the roles does not have enough privileges to cause a loss on its own, but collusion between them could lead to a loss.

8.3 Cross-Chain Staleness Thresholds Add Up

Note Version 1

The system applies two separate staleness checks when aggregating AUM across chains:

1. On each Spoke chain, `Caliber.getDetailedAum()` checks that every position was updated within `_positionStaleThreshold`.
2. In the Hub, the Machine applies a second check when receiving Wormhole responses, verifying that the response timestamp is within `_caliberStaleThreshold`.

Since the response timestamp reflects when `getDetailedAum()` was called, these thresholds accumulate. The effective maximum staleness of position data included in the Machine's global AUM can be up to: `positionStaleThreshold + caliberStaleThreshold`.

8.4 Deposits Are Not Paused Automatically

Note Version 1

The Machine's accounting can fail, leading to a stale share price. However, the Machine does not automatically pause deposits or withdrawals in this case. This means that users can still deposit/withdraw assets, even when the share price is stale, which could lead to receiving a better price than if the price was up to date.

Deposits and withdrawals are only disabled if the recovery mode is triggered by the Security Council. The Security Council should monitor the accounting status and ensure that recovery mode is triggered if the accounting becomes stale and deviates from the true value of the assets in the Machine.

The implementations of the Depositor and Redeemer contracts must also take this case into account.

8.5 Notes for Risk Managers

Note Version 1

This is a collection of behaviors that risk managers should be aware of when managing the risk parameters and allowed instructions of a Machine:

- Allowed instructions should always use or reset all approvals that they give. Leftover approvals can be dangerous.
- There should be at most one `FLASHLOAN_MANAGEMENT` instruction and exactly one `ACCOUNTING` instruction per position id in a caliber.
- `ACCOUNTING` instructions must be robust against read-only reentrancy. Any protocol that they are reading from must be checked for reentrancy vectors and it must be ensured that the accounting instruction reverts in case it is called from an inconsistent intermediate state. Accounting is permissionless, so anyone could reenter.
- `ACCOUNTING` instructions must be able to robustly price positions. Manipulation possibilities should be carefully considered for assets that are not trivial to value. This includes assets that are a claim on another asset, such as LP/vault share tokens, options, NFTs, etc. If an oracle is used, the markets that the oracle reads from must be sufficiently liquid.
- `ACCOUNTING` instructions should only be able to revert temporarily. If an accounting instruction reverts over an extended period, its value will become stale, leading to a stale share price for the entire machine. This will likely necessitate enabling recovery mode in order to disable deposits/withdrawals at an incorrect share price.
- `ACCOUNTING` instructions must never double-count funds. If there are two positions that deposit to the same contract, the accounting instructions must be able to clearly identify which funds belong to which position.
- `HARVEST` instructions must never be able to spend base tokens.
- Allowed instructions on third-party contracts that are upgradeable may change in behavior. All third-party contracts should be monitored for contract upgrades or important parameter changes. A contract upgrade can completely change the outcome of an allowed instruction.
- Each allowed instruction has a cooldown that limits how often it can be called. However, if there are two instructions that have similar behavior, they do not share a cooldown. As a result, it is important not to allow too many instructions with similar behavior, as each one can separately cause a loss without being restricted by the cooldown.
- Swaps from base tokens have a cooldown. If there is an instruction that includes a swap within it, this will have a separate cooldown. Additionally, swaps on different Calibers do not share a

cooldown. By bridging tokens to different chains, a compromised/malicious operator can incur multiple losses from swaps. This must be taken into account when configuring the maximum loss allowed for swaps.

- Any part of an instruction that is allowed to revert in the Weiroll VM and has inputs that are not fixed in the bitmap should be considered skippable by the operator (by supplying incorrectly sized calldata). As a result, partial reverts should likely not be allowed.
- Actions taken by the operator have a maximum loss enforced. However, this is only possible for losses that happen immediately. In case a collateral position is liquidated, this loss will occur in a separate transaction, not in an action taken by the operator. As a result, the loss from a liquidation cannot be bounded by a slippage check. A compromised/malicious operator can intentionally create positions that are very close to liquidation, causing losses to the Machine and potentially profiting from the liquidation. Allowed instructions should only be able to create collateral positions on sufficiently liquid collateral, where a liquidation would cause an acceptable loss.
- The maximum loss that can be caused by a compromised/malicious operator in a short time is the sum of all allowed losses. If there is enough time to bridge, it is the sum of all allowed losses across all chains. As a result, an increased number of instructions leads to a higher maximum loss. The allowed instruction set should be kept minimal, and unused instructions should be removed.

8.6 Operator Can Lose Money by Bridging Repeatedly

Note Version 1

The operator is semi-trusted. They should only be able to cause limited losses, even if they are compromised or malicious. E.g. for swaps, it is acceptable that the operator can lose some slippage, but there is a time limit on how often they can swap.

However, for bridging there is no time-based cooldown. As a result, the operator can lose up to `maxBridgeLossBps` as often as they are able to get filled by the chosen bridge. In Across, fills can happen within a short time frame. Looping bridges with maximum loss could rapidly cause losses if the `maxBridgeLossBps` is not set very tightly and the security council does not react immediately by enabling recovery mode. The total loss is limited by the total available liquidity of fillers in Across (or other enabled bridges).

If the operator also acts as the Across filler, they can turn the incurred bridge loss into a profit for themselves. They are likely able to outcompete other fillers, as they have information about the transaction before anyone else. However, filling in across requires providing the funds on the other chain upfront. Receiving the funds back takes at least 2 hours, as of the time of writing. As a result, the attacker will likely be capital-constrained, only able to extract up to `maxBridgeLossBps` of their available bridging capital, unless the security council takes more than 2 hours to react, allowing the attack to be repeated.

In summary, the operator can potentially cause large losses by repeatedly briding, if they do not act as Across filler. If they do act as filler, they can make a profit but are likely constrained to smaller briding amounts, depending on their available capital.

As a result of this, it should be ensured that the `maxBridgeLossBps` is set as tightly as possible and that monitoring is in place to detect when abnormally large bridging losses occur so that the security council can react.

8.7 Positions Can Become Unlinked

Note Version 2

In some scenarios, positions may become unlinked, even when they share the same `groupId`:

- **Initial unlink until first accounting:** Positions only become linked once they've each been managed at least once (i.e. once they have a non-zero `lastAccountingTime` and have been added to their `groupId` set). If a management or accounting instruction on position A were ever to create or modify position B before B's own first management, B remains unlinked and will not be stale-invalidated or required in the same batch.
- **Zero-crossings break the link:** When a position's `currentValue` reaches zero (e.g. through a `managePosition()` call or some other reason), it is removed from both the global `_positionIds` and `_positionIdGroups[groupId]`. If that position's on-chain balance later increases outside of a management call, it does not automatically rejoin the group; subsequent batch updates for the remaining members will ignore it until its next explicit management, at which point it is re-added.

8.8 Redemptions May Be Temporarily Unavailable

Note Version 1

When a user wants to redeem, they can do so by calling the Redeemer, which will then call the Machine to burn shares and withdraw assets. However, if the Machine does not have enough assets to cover the redemption, it will revert. This can happen, as the assets are deployed to positions of the Calibers. In order to process redemptions when there are insufficient assets held by the Machine itself, the operator must withdraw assets from positions and send them to the Machine. In some cases, the liquidity of the third-party protocols that the positions are deployed to may be limited, leading to delays in the withdrawal process. This means that redemptions may not be immediately available, and users may experience delays when trying to redeem their shares. The operator is trusted to deploy assets in a way that corresponds to the expected liquidity. If an operator refuses to move assets to the Machine, redemptions will not be possible until the operator is replaced.

The implementation of the Redeemer contract must handle the case where redemptions revert. Users should be aware of the limitations on redemption availability.

8.9 Sequencer Downtime

Note Version 1

The accounting of the Machine relies on periodic AUM updated from all Calibers. In case the sequencer of a Caliber's chain is down, the AUM cannot be updated. This will lead to the Machine share price becoming stale, likely requiring recovery mode to be enabled. Downtime of one chain will affect the entire Machine.

8.10 Share Price Arbitrage

Note Version 1

The share price is calculated as the ratio of the Machine's total assets to its total shares. However, the total assets will always be slightly out of date, as the accounting is only done periodically. Additionally, the total assets are calculated based on oracle prices, which can change over time. As a result, it is likely possible to predict future share prices before they are updated. If an attacker can predict the share price,

they could arbitrage it by minting shares before the price is updated and redeeming them after the price is updated.

The volatility of the share price will depend on the underlying assets and strategies of the Machine. E.g. if the Machine holds assets that are volatile with respect to the accounting asset, the share price will be more volatile. If the Machine only holds yield-bearing versions of the accounting asset, the share price will be fairly stable. In the volatile case, there is also the possibility of partial position updates. E.g. if one position has made a loss while another has made a profit, the profitable one could be accounted for, while a stale value is used for the loss-making position. This would lead to a temporary increase in the share price.

The Depositor and Redeemer contracts can be used to implement measures against this. They can implement whitelists to limit the set of attackers, and queues to prevent immediate minting and redeeming of shares. This turns a guaranteed profit into a probabilistic one, as the attacker would have to wait to sell their shares. It also introduces a cost of capital, as the attacker's funds would be locked up for a certain period of time. Note that queues do not fully mitigate share price manipulation, as the manipulation could take place at the exact moment that the queue determines the exit price.

8.11 Swap Module Only Supports exactIn Swaps

Note **Version 1**

In SwapModule, the `swap()` function only supports exactIn swaps. If an operator tries to use an exactOut swap, any excess input tokens will be stuck in the SwapModule.

8.12 Wormhole Queries Have No Finality

Note **Version 1**

Wormhole CCQ queries can be made permissionlessly for any block. This includes the newest block, which generally has no finality guarantees. This means that it is possible that data from a block is queried that is later "forked out" of the chain.

In the worst case, there could be a donation to a Caliber contract that is later reverted, leading to an inflated share price. However, this would require the attacker to be sure that the block will be reverted.

Whether this is possible depends on the consensus mechanism of the Caliber's chain. E.g. for a rollup with a centralized sequencer, only the sequencer could perform such an attack. The attack would be detectable and likely result in the sequencer being removed from the system, which carries a high cost for them.