Makina Foundation

Makina Finance

Security Assessment Report

Version: 2.1

August, 2025

Contents

Introduction

Sigma Prime was commercially engaged to perform a time-boxed security review of the Makina Foundation components in scope. The review focused solely on the security aspects of the Solidity implementation of the contract, though general recommendations and informational comments are also provided.

Disclaimer

Sigma Prime makes all effort but holds no responsibility for the findings of this security review. Sigma Prime does not provide any guarantees relating to the function of the components in scope. Sigma Prime makes no judgements on, or provides any security review, regarding the underlying business model or the individuals involved in the project.

Document Structure

The first section provides an overview of the functionality of the Makina Foundation components contained within the scope of the security review. A summary followed by a detailed review of the discovered vulnerabilities is then given which assigns each vulnerability a severity rating (see Vulnerability Severity Classification), an *open/closed/resolved* status and a recommendation. Additionally, findings which do not have direct security implications (but are potentially of interest) are marked as *informational*.

The appendix provides additional documentation, including the severity matrix used to classify vulnerabilities within the Makina Foundation components in scope.

Overview

Makina Finance is a non-custodial DeFi execution engine that enables users, including asset managers, AI agents, and funds, to deploy automated, risk-managed, multi-chain yield strategies through tokenized vaults. It enforces exposure limits, atomic execution, and cross-chain capital allocation, providing real-time risk controls and diversified exposure across DeFi protocols.

Security Assessment Summary

Scope

The review was conducted on the files hosted on the makina-core/ and makina-periphery/ repositories.

The scope of this time-boxed review was strictly limited to files at commit 922533f for `makina-core/` and 004374b for `makina-periphery/`.

Note: third party libraries and dependencies were excluded from the scope of this assessment.

Retesting activities have been performed on individual PRs for each of the relevant findings and also included PR #146 and PR #147.

Approach

The security assessment covered components written in Solidity.

The manual review focused on identifying issues associated with the business logic implementation of the contracts. This includes their internal interactions, intended functionality and correct implementation with respect to the underlying functionality of the Ethereum Virtual Machine (for example, verifying correct storage/memory layout).

Additionally, the manual review process focused on identifying vulnerabilities related to known Solidity anti-patterns and attack vectors, such as re-entrancy, front-running, integer overflow/underflow and correct visibility specifiers.

For a more detailed, but non-exhaustive list of examined vectors, see [1, 2].

To support the Solidity components of the review, the testing team may use the following automated testing tools:

- Aderyn: `https://github.com/Cyfrin/aderyn`
- Slither: `https://github.com/trailofbits/slither`
- Mythril: `https://github.com/ConsenSys/mythril`

Output for these automated tools is available upon request.

Coverage Limitations

Due to the time-boxed nature of this review, all documented vulnerabilities reflect best effort within the allotted, limited engagement time. As such, Sigma Prime recommends to further investigate areas of the code, and any related functionality, where majority of critical and high risk vulnerabilities were identified.

Findings Summary

The testing team identified a total of 15 issues during this assessment. Categorised by their severity:

- High: 1 issue.
- Medium: 3 issues.
- Low: 5 issues.
- Informational: 6 issues.

Detailed Findings

This section provides a detailed description of the vulnerabilities identified within the Makina Foundation components in scope. Each vulnerability has a severity classification which is determined from the likelihood and impact of each issue by the matrix given in the Appendix: Vulnerability Severity Classification.

A number of additional properties of the components, including optimisations, are also described in this section and are labelled as "informational".

Each vulnerability is also assigned a **status**:

- ***Open:*** the issue has not been addressed by the project team.
- ***Resolved:*** the issue was acknowledged by the project team and updates to the affected components(s) have been made to mitigate the related risk.
- ***Closed:*** the issue was acknowledged by the project team but no further actions have been taken.

Summary of Findings

ID	Description	Severity	Status
MAK-01	Malicious Operator Can Spoof `minOutputAmount` To Manipulate Share Price	High	**Resolved**
MAK-02	Improper Validation In Swap Flow Allows Malicious Operator To Steal Tokens	Medium	**Closed**
MAK-03	Invariant Violation Via Malicious Transfer Cancellation In Bridge Accounting	Medium	**Closed**
MAK-04	Potential Accounting Mismatch In Bridge Transfer Cancellation	Medium	**Closed**
MAK-05	Lack Of Enforcement On Virtual Offset May Weaken Inflation Attack Protection	Low	**Closed**
MAK-06	DAI Flash Loan Repayment's `forceApprove()` Breaks Atomic Transactions	Low	**Resolved**
MAK-07	No Staleness Check For `lastTotalAum`	Low	**Closed**
MAK-08	Any `state` Longer Than 128 Bytes Is Not Hashed	Low	**Closed**
MAK-09	Zero Flash Loan Fee Assumption In Balancer V2	Low	**Resolved**
MAK-10	Redundant Double Call In Position Management	Informational	**Closed**
MAK-11	Incorrect Decimal Unit Used In `previewDeposit` Calculation	Informational	**Closed**
MAK-12	Assumption Of Default Value For ERC20 Token Decimals	Informational	**Closed**
MAK-13	No Support For Tokens With Different Decimals On Different Chains	Informational	**Closed**
MAK-14	Assets With More Than 18 Decimals Are Not Supported	Informational	**Closed**
MAK-15	Usage of Solidity Version Newer Than `0.8.20`	Informational	**Closed**

MAK-01	Malicious Operator Can Spoof `minOutputAmount` To Manipulate Share Price		
Asset	`BridgeAdapter.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: High	Impact: High	Likelihood: Medium

Description

The Across bridge does not enforce message integrity. As such, a malicious operator can modify some of the fields in a cross-chain message, such as `minOutputAmount`, without invalidating the message.

This results in incorrect accounting and allows the operator to mint shares at a lower than intended share price. A possible attack is as follows:

1. Assume a fresh machine that has all its tokens currently idle in the machine.
2. The operator calls `machine.transferToSpokeCaliber()` to transfer all the tokens to a Spoke caliber.
3. The operator calls `authorizeInBridgeTransfer()` on the Spoke `caliberMailbox`, but modifies the message hash such that `msg.minOutputAmount` is zero.
4. The operator does not call `sendOutBridgeTransfer()` but rather sends their own message to the Spoke caliber mailbox via Across with only 1 `wei` of the `outputToken` and the message crafted in step 3. The Spoke bridge adapter accepts this message since the hashes match, and stores an `InBridgeTransfer` struct in `$._incomingTransfers` where `receivedAmount` is only 1 `wei` while `inputAmount` is the total amount of tokens sent in step 2.
5. The operator calls `claimInBridgeTransfer()` which updates the Spoke caliber's internal accounting, such that `caliberMailbox._bridgesIn == inputAmount` while the actual token balance of the mailbox is only 1 `wei`.
6. After this accounting data propagates to the hub, the total AUM will be very low since `spokeCaliberData.machineBridgesOut` is fully decremented by `spokeCaliberData.caliberBridgesIn` and `spokeCaliberData.netAum` is only 1 `wei`. As a result, shares can be minted at a very low price and potentially sold on a third party market.
7. Since the bridge state is now invalid, `resetBridgingState()` would have to be called manually to recover the actual tokens and restore the AUM and original share price.

```
function _receiveInBridgeTransfer(bytes memory encodedMessage, address receivedToken, uint256 receivedAmount)
    internal
{
    BridgeAdapterStorage storage $ = _getBridgeAdapterStorage();

    bytes32 messageHash = keccak256(encodedMessage);
    if (!$._expectedInMessages[messageHash]) {
        revert Errors.UnexpectedMessage();
    }
    delete $._expectedInMessages[messageHash];

    BridgeMessage memory message = abi.decode(encodedMessage, (BridgeMessage));

    if (message.destinationChainId != block.chainid) {
        revert Errors.InvalidRecipientChainId();
    }
    if (receivedToken != message.outputToken) {
        revert Errors.InvalidOutputToken();
    }
    if (receivedAmount < message.minOutputAmount) {
        revert Errors.InsufficientOutputAmount();
    }
    if (message.inputAmount < receivedAmount) {
        revert Errors.InvalidInputAmount();
    }

    uint256 id = $._nextInTransferId++;
    $._incomingTransfers[id] = InBridgeTransfer(
        message.sender,
        message.originChainId,
        message.inputToken,
        message.inputAmount,
        receivedToken,
        receivedAmount
    );
    _getSet($._pendingInTransferIds[receivedToken]).add(id);
    $._reservedBalances[receivedToken] += receivedAmount;
    emit InBridgeTransferReceived(id);
}
```

```
function authorizeInBridgeTransfer(bytes32 messageHash) external override onlyController {
    BridgeAdapterStorage storage $ = _getBridgeAdapterStorage();

    if ($._expectedInMessages[messageHash]) {
        revert Errors.MessageAlreadyAuthorized();
    }
    $._expectedInMessages[messageHash] = true;

    emit InBridgeTransferAuthorized(messageHash);
}
```

Note that a malicious operator can also modify other fields in the cross-chain message, such as `outTransferId`, `sender`, `recipient`, `originChainId` and `inputToken`. However, the testing team found no direct impact from manipulating these fields.

Recommendations

Verify that `receivedAmount` does not deviate from `inputAmount` by more than `maxBridgeLossBps` on the receiving end.

Resolution

The development team has resolved the issue in PR #144 by verifying that the value loss does not exceed `maxBridgeLossBps` on incoming transfers.

MAK-02	Improper Validation In Swap Flow Allows Malicious Operator To Steal Tokens		
Asset	`SwapModule.sol`		
Status	**Closed:** See Resolution		
Rating	Severity: Medium	Impact: Medium	Likelihood: Medium

Description

The `swap()` function allows an operator to pass arbitrary `data` within the `ISwapModule.SwapOrder` structure:

```
function swap(ISwapModule.SwapOrder calldata order) external override nonReentrant onlyOperator {
    _swap(order);
}
```

```
struct SwapOrder {
    uint16 swapperId;
    bytes data;
    address inputToken;
    address outputToken;
    uint256 inputAmount;
    uint256 minOutputAmount;
}
```

A malicious operator can exploit this mechanism to steal non-base tokens using the following steps:

1. **Arbitrary Call Injection**: The internal `_swap()` function forwards the arbitrary `data` to an external contract (e.g., `OdosRouterV2`) via a low-level `.call()`:

   ```
   (bool success,) = executionTarget.call(order.data);
   ```

   Both `approvalTarget` and `executionTarget` are fetched from trusted storage by `swapperId`, and currently point to:

   ```
   {
     "approvalTarget": "0xCf5540fFFCdC3d510B18bFcA6d2b9987b0772559",
     "executionTarget": "0xCf5540fFFCdC3d510B18bFcA6d2b9987b0772559"
   }
   ```

2. **Forged Swap Call**: Since `order.data` is operator-controlled, they can craft a payload calling `OdosRouterV2._swapApproval()` with:

   - `tokenInfo.inputReceiver = operator address`
   - `tokenInfo.outputReceiver = operator address`
   - `executor = malicious contract`
   - Arbitrary `outputQuote` and `outputMin` to bypass validations

3. **Fund Redirection**: The malicious payload causes the non-base tokens to be sent to the operator instead of a legitimate receiver `SwapModule`. Since the token involved is a non-base token, the slippage protection mechanism fails to detect this issue.

Consequently, the malicious operator can craft the data to divert the non-base tokens to their own address.

Recommendations

Add strict validation on the structure and contents of `order.data` before passing it to the `executionTarget`.

Resolution

The development team has closed the issue with the following rationale:

> *"The vulnerability only affects potential rewards and not user funds. Besides, standardized validation of `order.data` is not practically feasible due to its arbitrary and application-specific structure. However, off-chain monitoring will be implemented to detect any operator deviations related to reward swapping. If such deviations are identified, appropriate actions will be taken by the DAO."*

MAK-03	Invariant Violation Via Malicious Transfer Cancellation In Bridge Accounting		
Asset	`BridgeAdapter.sol`		
Status	**Closed:** See Resolution		
Rating	Severity: Medium	Impact: Medium	Likelihood: Medium

Description

There is an invariant in the bridging logic that the amount bridged out from the Machine (`mOut`) must be greater than or equal to the amount bridged into the Spoke Caliber (`cIn`).

```
if (mOut > cIn) {
    revert Errors.PendingBridgeTransfer();
} else if (mOut < cIn) {
    revert Errors.BridgeStateMismatch();
}
```

However, a malicious operator can break this invariant through the following sequence:

1. The operator initiates a bridge-out transfer of X tokens from the Machine using `sendOutBridgeTransfer()` , which increases `mOut` to X.
2. The Spoke Caliber receives the transfer and sets `bridgeIn (cIn)` to X.
3. The operator then directly transfers X tokens to the `BridgeAdapter` contract on the Hub chain and calls `cancelOutBridgeTransfer()` using the same transfer Id, even though it was already successfully bridged. This incorrectly decreases `mOut` back to 0.
4. Now, when the Machine queries the Spoke Caliber's state, it sees `mOut = 0` and `cIn = X` , violating the invariant (`mOut < cIn`).

This causes a state inconsistency that requires manual intervention by an authority via `resetBridgingState()` to restore correct accounting.

Importantly, the operator does not need to spend a large amount of funds, this attack can be executed using a small token amount.

Recommendations

To prevent such inconsistencies, avoid relying on `IERC20.balanceOf()` to determine token availability for cancellation. Instead, maintain an internal accounting system to track balances and ensure transfer cancellations are only allowed for genuinely unprocessed or refunded transfers.

Resolution

The development team has closed the issue with the following rationale:

"The `mOut < cIn` is not an invariant, but a safeguard against operator deviations that could lead to share price manipulation. The scenario described is actually the one for which we added this check, and authority intervention is the intended process in such case."

MAK-04	Potential Accounting Mismatch In Bridge Transfer Cancellation		
Asset	`BridgeAdapter.sol`		
Status	**Closed:** See Resolution		
Rating	Severity: Medium	Impact: Medium	Likelihood: Medium

Description

Operators can cancel the wrong transfer ID after a refund, causing internal accounting drift and leaving tokens stuck in the `BridgeAdapter`.

Operators are allowed to cancel bridge-out transfers, either for tokens that are scheduled to be bridged or for those refunded by Across (e.g., due to failed intents). If a token was sent out and later refunded, the cancellation process checks whether the `BridgeAdapter` has enough balance to recover the tokens. If so, it updates the `bridgeOut` accounting to reflect the change:

```
if (_getSet($._sentOutTransferIds[receipt.inputToken]).remove(id)) {
    if (
        IERC20(receipt.inputToken).balanceOf(address(this)) <
        $._reservedBalances[receipt.inputToken] + receipt.inputAmount
    ) {
        revert Errors.InsufficientBalance();
    }
}
```

Later in the flow, if the transfer is flagged as refunded, the `bridgeOut` is reduced accordingly:

```
if (refund) {
    uint256 bridgeOut = $._bridgesOut.get(token);
    $._bridgesOut.set(token, bridgeOut - inputAmount);
}
```

However, this logic can lead to an accounting issue if an operator, either by mistake or maliciously, cancels the wrong transfer ID. For example:

1. **Initial Bridging**:
   - 100 tokens are bridged out.
   - Later, another 70 tokens are also bridged out.
   - Total `bridgeOut = 170` on the Spoke Caliber.
2. **Bridge Outcomes**:
   - Across refunds the first 100 tokens due to a failure.
   - Across successfully bridges the 70 tokens.
3. **Operator's Action**:
   - Instead of cancelling the transfer id corresponding to the refunded 100 tokens, the operator cancels the one for the successfully bridged 70 tokens.
4. **Execution Flow**:

- `bridgeOut` is updated from 170 → 100.
- The contract has 100 tokens (from the refund), which satisfies the balance check during cancellation.
- 70 tokens are returned to the Spoke Caliber, although they had already arrived at the Machine.
- 30 tokens are locked in the `BridgeAdapter`.

5. **Resulting State**:

   - On the Machine: `mIn = 70`, `cOut = 100`
   - Balances:
     - Machine: 70
     - Spoke Caliber: 70
     - BridgeAdapter: 30
   - The total AUM is correct, but internal accounting (`mIn`, `cOut`) is inconsistent.
   - Those 30 tokens are effectively "stuck" in the BridgeAdapter, requiring to call `resetBridgingState` manually.

Recommendations

The cancellation logic should validate that the transfer ID corresponds to the refunded token and amount, ideally by querying Across to ensure the refund actually applies to that transfer. This would ensure proper reconciliation and avoid incorrect fund reallocation or accounting drift.

Resolution

The development team has closed the issue with the following rationale:

> *"Authority intervention is the standard process. This design choice was made because it is not possible to query Across to verify that a refund corresponds to a specific transfer."*

MAK-05	Lack Of Enforcement On Virtual Offset May Weaken Inflation Attack Protection		
Asset	`Machine.sol`		
Status	**Closed:** See Resolution		
Rating	Severity: Low	Impact: Low	Likelihood: Low

Description

The inflation protection mechanism can fail if the virtual offset is zero, leaving the system partially exposed to front-run donation attacks.

The system mitigates inflation attacks using virtual asset and offset logic:

```
function convertToShares(uint256 assets) public view override returns (uint256) {
    MachineStorage storage $ = _getMachineStorage();
    return assets.mulDiv(
        IERC20($._shareToken).totalSupply() + 10 ** $._shareTokenDecimalsOffset,
        $._lastTotalAum + 1
    );
}
```

However, there is no enforcement to ensure that `_shareTokenDecimalsOffset` is non-zero. When `atDecimals` equals 18, the offset becomes zero, nullifying its protective effect.

With a zero offset, attackers cannot profit immediately from front-running donations but can still gain an advantage as more users deposit over time, eroding fairness and increasing systemic risk.

Recommendations

Although slippage protection offers some defense, it is recommended to enforce a minimum virtual offset when `$._shareTokenDecimalsOffset` equals zero to better guard against delayed inflation attacks.

Resolution

The development team has closed the issue with the following rationale:

> *"For simplicity, and as slippage protection is implemented, we chose to keep the current implementation. As mentioned by OpenZeppelin, in their ERC4626 implementation that integrates a similar mechanism:*
>
> *While not fully preventing the attack, analysis shows that the default offset (0) makes it non-profitable even if an attacker is able to capture value from multiple user deposits, as a result of the value being captured by the virtual shares (out of the attacker's donation) matching the attacker's expected gains. With a larger offset, the attack becomes orders of magnitude more expensive than it is profitable."*

MAK-06	DAI Flash Loan Repayment's `forceApprove()` Breaks Atomic Transactions		
Asset	`FlashloanAggregator.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: Low	Impact: Low	Likelihood: Low

Description

Using `forceApprove()` during flash loan repayment introduces a race condition that breaks multi-step transaction atomicity and invalidates prior approvals.

The repayment logic calls `IERC20(token).forceApprove(msg.sender, amount);` to reset DAI allowance to the exact loaned amount. This action overrides any pre-existing approvals without considering their context:

```
function onFlashLoan(address initiator, address token, uint256 amount, uint256 fee, bytes calldata data)
    external
    returns (bytes32)
{
    // ...
    IERC20(token).forceApprove(msg.sender, amount); // Resets approval to exact amount
    // ...
}
```

If a flash loan occurs between steps of a multi-operation sequence, the `forceApprove()` call zeroes previous allowances. Subsequent operations that rely on these approvals fail unexpectedly, disrupting transaction flow and potentially causing state inconsistencies.

Consider the following scenario:

1. The protocol first approves MakerDAO for a specific operation
2. A flash loan occurs between these operations
3. The repayment's `forceApprove()` resets all prior approvals to zero
4. Subsequent operations assume this approval remains valid and fail unexpectedly

Recommendations

Replace `forceApprove()` with an incremental approval pattern:

```
// Safe approval alternative
IERC20(token).safeIncreaseAllowance(msg.sender, amount);
```

Resolution

This issue was fixed in PR #12 by replacing `forceApprove()` with `safeIncreaseAllowance()`

MAK-07	No Staleness Check For `lastTotalAum`		
Asset	`Machine.sol`		
Status	**Closed:** See Resolution		
Rating	Severity: Low	Impact: Medium	Likelihood: Low

Description

The system may use an outdated share price during deposits or redemptions, leading to inaccurate asset valuations.

The `lastTotalAum` state variable, which represents the amount of assets under management and determines share price, is only updated through the `updateTotalAum()` function. However, its freshness is not validated when executing `deposit()` or `redeem()` operations.

If `lastTotalAum` is stale, share price calculations may be based on outdated data, resulting in incorrect valuation of user deposits and redemptions:

```
function convertToShares(uint256 assets) public view override returns (uint256) {
    MachineStorage storage $ = _getMachineStorage();
    return
        assets.mulDiv(IERC20($._shareToken).totalSupply() + 10 ** $._shareTokenDecimalsOffset, $._lastTotalAum + 1);
}
```

Recommendations

Consider using the already implemented `lastGlobalAccountingTime` to ensure `lastTotalAum` is not stale during deposits and redemptions.

Resolution

The development team has closed the issue with the following rationale:

> *"While the entire accounting flow is permissionless and can be executed by anyone, the protocol will also incorporate off-chain monitoring to detect prolonged operator inactivity. In such cases, the Security Council may trigger recovery mode, disabling both deposits and redemptions."*

MAK-08	Any `state` Longer Than 128 Bytes Is Not Hashed		
Asset	`Caliber.sol`		
Status	**Closed:** See Resolution		
Rating	Severity: Low	Impact: Medium	Likelihood: Low

Description

State bytes beyond the 128th position are not included in the Merkle-constrained hash, allowing operators to arbitrarily modify them.

The `_getStateHash()` function uses a bitmap to determine which state bytes are hashed for Merkle proof verification. However, the bitmap only covers the first 128 bytes, resulting in all bytes beyond this range being excluded from the hash.

Any state data after the 128th byte remains unconstrained by the Merkle root, enabling an operator to arbitrarily alter these values without breaking the proof.

```
function _getStateHash(bytes[] calldata state, uint128 bitmap) internal pure returns (bytes32) {
    if (bitmap == uint128(0)) {
        return bytes32(0);
    }

    uint8 i;
    bytes memory hashInput;

    // Iterate through the state and hash values corresponding to indices marked in the bitmap.
    for (i; i < state.length; ++i) {
        // If the bit is set as 1, hash the state value.
        if (bitmap & (0x80000000000000000000000000000000 >> i) != 0) { //@audit always false for `i >= 128`
            hashInput = bytes.concat(hashInput, keccak256(state[i]));
        }
    }
    return keccak256(hashInput);
}
```

Recommendations

If support for `state` larger than 128 bytes is deemed necessary, consider increasing the size of the `bitmap`. Otherwise, consider adding a check to ensure `state` is smaller or equal to 128 bytes.

Resolution

The development team has closed the issue with the following rationale:

> *"The implementation of Enso-Weiroll that we use here limits the state size to 128."*

MAK-09	Zero Flash Loan Fee Assumption In Balancer V2		
Asset	`FlashloanAggregator.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: Low	Impact: Low	Likelihood: Low

Description

The code incorrectly assumes Balancer V2 flash loan fees are always zero, leaving it vulnerable if the fee is increased by the protocol owner.

When obtaining a flash loan from Balancer V2, the code currently assumes the fee will always be zero, as shown below:

```
/// @inheritdoc BalancerV2FlashloanRecipient
function receiveFlashLoan(
    BalancerIERC20[] memory tokens,
    uint256[] memory amounts,
    uint256[] memory feeAmounts,
    bytes memory userData
) external {
    // ....

    // Ensure no fees are charged
    if (feeAmounts[0] != 0) {
        revert InvalidFeeAmount();
    }

    // ....
}
```

However, this assumption is not always correct because Balancer V2's owner can adjust the flash loan fee using the `setFlashLoanFeePercentage()` function:

```
function setFlashLoanFeePercentage(uint256 newFlashLoanFeePercentage) external authenticate {
    _require(
        newFlashLoanFeePercentage <= _MAX_PROTOCOL_FLASH_LOAN_FEE_PERCENTAGE,
        Errors.FLASH_LOAN_FEE_PERCENTAGE_TOO_HIGH
    );
    _flashLoanFeePercentage = newFlashLoanFeePercentage;
    emit FlashLoanFeePercentageChanged(newFlashLoanFeePercentage);
}
```

Recommendations

To make the code more robust, it is advised to fetch the current flash loan fee from Balancer V2 rather than assuming it will remain zero.

Resolution

This issue was fixed in PR #13 by adding support for non-zero fees.

MAK-10	Redundant Double Call In Position Management
Asset	`Caliber.sol`
Status	**Closed:** See Resolution
Rating	Informational

Description

`_managePosition()` calls `_accountForPosition()` twice unnecessarily, wasting gas and reducing efficiency.

The first call to `_accountForPosition(acctInstruction, true);` is used solely to validate the instruction, ignoring return values. The second call `(uint256 value, int256 change) = _accountForPosition(acctInstruction, false);` retrieves the required outputs but skips validation. This duplication could be avoided by combining validation and return logic in a single call.

Unnecessary repeated calls increase gas consumption, making the function less efficient and more expensive for users.

Recommendations

Remove the first call to the function and change the second call to:

```
(uint256 value, int256 change) = _accountForPosition(acctInstruction, true);
```

This will validate the instruction, account for position at correct time and save the computation cost of accounting for position without the need for values.

Resolution

The development team has closed the issue with the following rationale:

> *"Removing the first call to `_accountForPosition()` would introduce a high vulnerability here, which would basically allow to circumvent value loss check. Consider the following scenario:*
>
> *1. A position is open in the caliber, and worth 1000 usd. `_maxPositionIncreaseLossBps` is set to 1%.*
> *2. The position value increases by 10% due to positive yield, but this increase is not yet accounted for.*
> *3. A management of this positions incurs a 2% loss.*
> *4. The 2% loss is not detected, because the result of `_accountForPosition()` is interpreted as an 8% value increase."*

MAK-11	Incorrect Decimal Unit Used In `previewDeposit` Calculation
Asset	`PreDepositVault.sol`
Status	**Closed:** See Resolution
Rating	Informational

Description

The `previewDeposit` function misuses the deposit token's decimal unit in share price calculation, breaking the intended inflation protection logic.

When `previewDeposit` is called, the deposit amount is converted into the accounting token, which governs the actual deposit logic. However, the calculation below incorrectly uses `dtUnit`, which reflects the deposit token's decimal precision rather than the accounting token:

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

This misalignment can distort share pricing and undermine the inflation protection mechanism.

Recommendations

Replace `dtUnit` with `1` in the calculation to align with the accounting token's logic and preserve inflation protection.

Resolution

The development team has closed this issue as it is a non-issue, providing the following clarification:

> *The detail of the calculation in `previewDeposit` goes as follows:*
>
> ```
> shares = assetsExpressedInAT * (stSupply + 10 ** $._shareTokenDecimalsOffset) / (dtBalExpressedInAT + 1)
> = (assets * price_d_a / dtUnit) * (stSupply + 10 ** $._shareTokenDecimalsOffset) / ((dtBal * price_d_a / dtUnit) + 1)
> = (assets * price_d_a) * (stSupply + 10 ** $._shareTokenDecimalsOffset) / ((dtBal * price_d_a) + dtUnit)
> ```

MAK-12	Assumption Of Default Value For ERC20 Token Decimals
Asset	`DecimalsUtils.sol`
Status	**Closed:** See Resolution
Rating	Informational

Description

Assuming 18 decimals for tokens without an implemented `decimals()` function introduces risk of mispricing and incorrect value calculations:

```
function _getDecimals(address asset_) internal view returns (uint8) {
    (bool success, bytes memory encodedDecimals) =
        address(asset_).staticcall(abi.encodeCall(IERC20Metadata.decimals, ()));
    if (success && encodedDecimals.length >= 32) {
        uint256 returnedDecimals = abi.decode(encodedDecimals, (uint256));
        if (returnedDecimals <= type(uint8).max) {
            return uint8(returnedDecimals);
        }
    }
    return DEFAULT_DECIMALS; // Currently set to 18
}
```

The `decimals()` function in ERC20 is optional. When it is missing, the protocol defaults to 18 decimals. This implicit assumption is flawed because there is no guarantee the token follows this convention.

Defaulting to 18 decimals can cause inaccurate calculations and misrepresentation of token values, potentially leading to financial discrepancies or exploitation.

Recommendations

Set `DEFAULT_DECIMALS` to `0` to avoid false assumptions and ensure more conservative and accurate handling of tokens lacking a defined decimal value.

Resolution

The development team has closed the issue with the following rationale:

> *"We opted to support tokens that may not implement `decimals()`. A value of 0 decimals is generally unrealistic, such tokens are more likely to implicitly follow the 18-decimals convention. Furthermore 0 decimals would not pass our check on minimal decimals. This behaviour also mirrors OpenZeppelin's approach, with their ERC-4626 implementation that defaults to 18 decimals.*
>
> *Note that token listing is permissioned and controlled by the DAO, which is expected to perform due diligence before adding any token."*

MAK-13	No Support For Tokens With Different Decimals On Different Chains
Asset	`BridgeController.sol`
Status	**Closed:** See Resolution
Rating	Informational

Description

The bridge transfer validation fails to account for token decimal differences, blocking cross-chain transfers with mismatched decimal formats.

In `_scheduleOutBridgeTransfer()`, the `minOutputAmount` must not deviate from `inputAmount` by more than `maxBridgeLossBps`. However, this check assumes both tokens share the same decimal precision, which is not always true across chains.

```
function _scheduleOutBridgeTransfer(
    uint16 bridgeId,
    uint256 destinationChainId,
    address recipient,
    address inputToken,
    uint256 inputAmount,
    address outputToken,
    uint256 minOutputAmount
) internal {
    BridgeControllerStorage storage $ = _getBridgeControllerStorage();
    address adapter = getBridgeAdapter(bridgeId);
    if (!$._isOutTransferEnabled[bridgeId]) {
        revert Errors.OutTransferDisabled();
    }
    if (minOutputAmount < inputAmount.mulDiv(MAX_BPS - $._maxBridgeLossBps[bridgeId], MAX_BPS)) {
        revert Errors.MaxValueLossExceeded();
    } //@audit does not work if tokens have different decimals
    if (minOutputAmount > inputAmount) {
        revert Errors.MinOutputAmountExceedsInputAmount();
    }
    IERC20(inputToken).forceApprove(adapter, inputAmount);
    IBridgeAdapter(adapter).scheduleOutBridgeTransfer(
        destinationChainId, recipient, inputToken, inputAmount, outputToken, minOutputAmount
    );
}
```

Tokens with differing decimal places on source and destination chains cannot be bridged, reducing interoperability and potentially breaking functionality.

Recommendations

Incorporate decimal normalisation in the validation logic to correctly compare amounts across tokens with different precisions.

Resolution

The development team has closed the issue with the following rationale:

> *"As outlined in the Liquidity Bridging section of SPECIFICATIONS.md, the protocol assumes that the input and output tokens involved in a bridge transfer are homologous and share the same number of decimals. Hence Tokens with different decimals on different chains are hence not supported."*

MAK-14	Assets With More Than 18 Decimals Are Not Supported
Asset	`Machine.sol`
Status	**Closed:** See Resolution
Rating	Informational

Description

The protocol does not support accounting tokens with more than 18 decimals.

For example, an accounting token with 24 decimals will cause an overflow in the initializer of `Machine` and cause execution to revert:

```
function initialize(
    MachineInitParams calldata mParams,
    MakinaGovernableInitParams calldata mgParams,
    address _preDepositVault,
    address _shareToken,
    address _accountingToken,
    address _hubCaliber
) external override initializer {
    // ...

    uint256 atDecimals = DecimalsUtils._getDecimals(_accountingToken);

    // ...
    $._shareTokenDecimalsOffset = DecimalsUtils.SHARE_TOKEN_DECIMALS - atDecimals; //@audit overflow if `atDecimals > 18`
```

Recommendations

Consider modifying the implementation such that assets with more than 18 decimals are supported.

Resolution

The development team has closed the issue with the following rationale:

> *"Before a token can be used as an accounting token or base token, it must be registered in the* `OracleRegistry` *via* `setFeedRoute()`*. This function performs checks on token decimals and enforces a decimals range between* `MIN_DECIMALS = 6` *and* `MAX_DECIMALS = 18`*.*
>
> *Consequently, the protocol explicitly does not support assets with fewer than 6 or more than 18 decimals."*

MAK-15	Usage of Solidity Version Newer Than `0.8.20`
Asset	`*.sol`
Status	**Closed:** See Resolution
Rating	Informational

Description

The contracts rely on new EVM opcodes that may not be supported on all rollups, creating potential deployment failures or runtime issues.

The contracts are compiled with Solidity 0.8.28, which introduces opcodes such as `PUSH0` and `MCOPY`. Additionally, the `FlashloanAggregator` uses transient storage, invoking `TSTORE` and `TLOAD`. Some rollups (e.g., Linea) do not yet support these opcodes.

Deployments on unsupported rollups will fail or behave unpredictably, limiting compatibility across target chains.

Recommendations

Verify opcode support on every intended deployment chain.

If unsupported, configure the Solidity compiler to target a lower EVM target version that omits these opcodes.

Resolution

The development team has closed the issue with the following rationale:

> *"Opcode support will be verified for any deployment chain."*

Appendix A Vulnerability Severity Classification

This security review classifies vulnerabilities based on their potential impact and likelihood of occurance. The total severity of a vulnerability is derived from these two metrics based on the following matrix.

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

Table 1: Severity Matrix - How the severity of a vulnerability is given based on the *impact* and the *likelihood* of a vulnerability.

References

[1] Sigma Prime. Solidity Security. Blog, 2018, Available: `https://blog.sigmaprime.io/solidity-security.html`. [Accessed 2018].

[2] NCC Group. DASP - Top 10. Website, 2018, Available: `http://www.dasp.co/`. [Accessed 2018].