Metamask Delegation Framework April 2025

Date	April 2025
Auditors	Vladislav Yaroshuk, George Kobakhidze

1 Executive Summary

This report presents the results of our engagement with **Metamask** to review updates to the **Metamask Delegation Framework**.

The review was conducted over two weeks, from **April 14, 2025** to **April 23, 2025**, by **Vladislav Yaroshuk** and **George Kobakhidze**.

This audit is a continuation of a series of audits that Diligence has done for the Metamask Delegation Framework and builds upon them. In particular, the last audit was done on commit `4779e62e00731acff24bffaeb942a52da0bc77b7`, and relevant updates to the code were compared to this commit. For this specific engagement, the focus was on a new enforcer for delegations called `MultiTokenPeriodEnforcer` and on the changes made to the `DelegationMetaSwapAdapter` contract.

The new `MultiTokenPeriodEnforcer` allows for delegations for either native tokens (such as ETH) and ERC-20 tokens. Specifically, it allows to delegate transfers based on transfer periods with associated amounts, tracking every period expenditure within the contract.

On the other hand, the `DelegationMetaSwapAdapter` is not a delegation enforcer. Instead, this contract facilitates delegations that have to do with swaps via aggregators such as the Metamask swap aggregator. The contract updates in scope of this audit focused on creating additional checks on the aggregator swap data. Namely, there is now a signature from the Metamask swap aggregator that verifies that the swap data is correct and not expired.

No critical design issues were discovered during the audit. However, due to the nature of quickly rising complexity with arbitrary delegated actions, we found a few edge cases that could result in significant unintended impacts. Additionally, a few more checks should be introduced to err on the side of caution and security.

2 Scope

Our review focused on the commit hash 0f8e128adebc45f81c7c3d5e35124450767a454d. The list of files in scope can be found in the Appendix.

2.1 Objectives

Together with the **Metamask** team, we identified the following priorities for our review:

1. Correctness of the implementation, consistent with the intended functionality and without unintended edge cases.
2. Identify known vulnerabilities particular to smart contract systems, as outlined in our Smart Contract Best Practices, and the Smart Contract Weakness Classification Registry.

3 Security Specification

This section describes, **from a security perspective**, the expected behavior of the system under audit. It is not a substitute for documentation. The purpose of this section is to identify specific security properties that were validated by the audit team.

3.1 Actors

The relevant actors are listed below with their respective abilities:

Delegators

Delegators are the users who allow delegates to act on their behalf. They do not have significant control or any administrative privileges within the system, outside of configuring their own delegations.

Delegates

On the other hand, delegates are the users who act within the system and perform contract calls. In this audit's context, delegates facilitate token transfers and swaps. They are limited by the `MultiTokenPeriodEnforcer` delegation enforcer, which restricts the amount they can transfer within a specific period, reducing trust assumptions. In the context of the `DelegationMetaSwapAdapter`, they are also limited in how quickly (or rather slowly) they can perform swaps with some additional checks for prices, as this data is signed by the Metamask aggregator.

`DelegationMetaSwapAdapter` Contract Owners.

The contract owners of `DelegationMetaSwapAdapter` have the ability to change some critical properties. That includes administrative actions like setting the address of the aggregator against which signed data will be verified.

Metamask Swap Aggregator

The aggregator address, or an address designated as the signer for it, is responsible for providing signed data for swaps with critical information such as price and expiration timestamp.

3.2 Trust Model

In any system, it's important to identify what trust is expected/required between various actors. For this audit, we established the following trust model:

Delegators

Delegators are not trusted to perform any critical operations beyond allowing delegates to act on their behalf. The system assumes that delegators will not perform any malicious activities but does not rely on them for any significant security properties.

Delegates

Delegates are trusted to facilitate transfers and perform swaps appropriately. As mentioned above, their actions are still limited by delegations, enforcers, and additional checks present in contracts like the `DelegationMetaSwapAdapter`. Still, the trust placed in delegates is significant, and abuse of delegations is possible.

`DelegationMetaSwapAdapter` Contract Owners

Contract owners have significant control over critical properties of the contracts, including setting the address signing for aggregators and configuring the aggregators. So, they are trusted to manage administrative items on the `DelegationMetaSwapAdapter` responsibly.

Metamask Swap Aggregator

While only doing one thing - signing swap data - that is a critical action that moves funds in the `DelegationMetaSwapAdapter` contract. So this entity is trusted to provide correct prices with short enough expirations so that the signature can't be used to abuse the delegator's funds.

4 Findings

Each issue has an assigned severity:

- Minor issues are subjective in nature. They are typically suggestions around best practices or readability. Code maintainers should use their own judgment as to whether to address such issues.
- Medium issues are objective in nature but are not security vulnerabilities. These should be addressed unless there is a clear reason not to.
- Major issues are security vulnerabilities that may not be directly exploitable or may require certain conditions in order to be exploited. All major issues should be addressed.
- Critical issues are directly exploitable security vulnerabilities that need to be fixed.

4.1 Missing Safe Transfer Validation in Execution Logic Major Acknowledged

Resolution

Acknowledged by the client with the following note:

> We added a warning in the documentation in this hash: bfca5f9163eb4c5b1a1ff309a5fc9d6a5815f538

Description

In the `beforeHook` function of the `MultiTokenPeriodEnforcer` contract, there is a validation that the mode of execution is `CALLTYPE_SINGLE` and `EXECTYPE_DEFAULT`, so the ERC-20 token transfers will always be executed via `_execute` function, which performs a low-level `call`. However, the function lacks critical safety checks typically found in secure token transfers. There is no validation that the token transfer succeeded by returning `true`, as required by the ERC-20 standard. This omission could result in silent failures when interacting with non-standard token contracts, allowing tokens that don't revert on failures and just return `false` to be incorrectly treated as successfully transferred.

Examples

src/enforcers/MultiTokenPeriodEnforcer.sol:L131-L144

```
function beforeHook(
    bytes calldata _terms,
    bytes calldata,
    ModeCode _mode,
    bytes calldata _executionCallData,
    bytes32 _delegationHash,
    address,
    address _redeemer
)
    public
    override
    onlySingleCallTypeMode(_mode)
    onlyDefaultExecutionMode(_mode)
{
```

src/DeleGatorCore.sol:L182-L216

```
 * @notice Executes an Execution from this contract
 * @dev Related: @erc7579/MSAAdvanced.sol
 * @dev This method is intended to be called through a UserOp which ensures the invoker has sufficient permissions
 * @param _mode The ModeCode for the execution
 * @param _executionCalldata The calldata for the execution
 */
function execute(ModeCode _mode, bytes calldata _executionCalldata) external payable onlyEntryPoint {
    (CallType callType_, ExecType execType_,,) = _mode.decode();

    // Check if calltype is batch or single
    if (callType_ == CALLTYPE_BATCH) {
        // destructure executionCallData according to batched exec
        Execution[] calldata executions_ = _executionCalldata.decodeBatch();
        // Check if execType is revert or try
        if (execType_ == EXECTYPE_DEFAULT) _execute(executions_);
        else if (execType_ == EXECTYPE_TRY) _tryExecute(executions_);
        else revert UnsupportedExecType(execType_);
    } else if (callType_ == CALLTYPE_SINGLE) {
        // Destructure executionCallData according to single exec
        (address target_, uint256 value_, bytes calldata callData_) = _executionCalldata.decodeSingle();
        // Check if execType is revert or try
        if (execType_ == EXECTYPE_DEFAULT) {
            _execute(target_, value_, callData_);
        } else if (execType_ == EXECTYPE_TRY) {
            bytes[] memory returnData_ = new bytes[](1);
            bool success_;
            (success_, returnData_[0]) = _tryExecute(target_, value_, callData_);
            if (!success_) emit TryExecuteUnsuccessful(0, returnData_[0]);
        } else {
            revert UnsupportedExecType(execType_);
        }
    } else {
        revert UnsupportedCallType(callType_);
    }
}
```

```
// from erc7579-implementation/src/core/ExecutionHelper.sol

    function _execute(
        address target,
        uint256 value,
        bytes calldata callData
    )
        internal
        virtual
        returns (bytes memory result)
    {
        /// @solidity memory-safe-assembly
        assembly {
            result := mload(0x40)
            calldatacopy(result, callData.offset, callData.length)
            if iszero(call(gas(), target, value, result, callData.length, codesize(), 0x00)) {
                // Bubble up the revert if the call reverts.
                returndatacopy(result, 0x00, returndatasize())
                revert(result, returndatasize())
            }
            mstore(result, returndatasize()) // Store the length.
            let o := add(result, 0x20)
            returndatacopy(o, 0x00, returndatasize()) // Copy the returndata.
            mstore(0x40, add(o, returndatasize())) // Allocate the memory.
        }
    }
```

Recommendation

We recommend creating a separate execution logic for ERC-20 token transfers, which will validate all the edge case scenarios:

- Even though with the current code a call to EOA will revert, we still recommend verifying that the target address is a contract before executing the low-level `call`, e.g., using `Address.isContract`.
- In cases when the token returns a variable, checking that the token transfer returns `true` or does not return `false`, in line with the ERC-20 specification, as it is done in the SafeERC20 library.

4.2 `getTermsInfo` Reverts Because of Block Gas Limit Major ✓ Fixed

Resolution

Fixed in 49b57f4a55f10d83fe9b1a990a0dd52cced186c8 and PR 104 by changing how token information gets retrieved in `getTermsInfo()`. In particular, a `uint256 _tokenIndex` argument was added to access specific token configurations immediately as opposed to searching through the whole configuration set.

Description

The `getTermsInfo` function decodes a flattened `bytes` array containing multiple fixed-size token configurations. Each record is 116 bytes long and includes a token address, `periodAmount`, `periodDuration`, and `startDate`. The function iterates over all entries to find a match for the provided `_token` address and then extracts the associated parameters.

However, the function always iterates over the entire `terms` array, which makes this approach gas-inefficient, particularly for larger arrays. Considering the PR 81, the system should be able to operate with large quantities of tokens, including 100 tokens. However, because the `MultiTokenPeriodEnforcer` enforcer will be used together with other enforcers and delegations, and the `redeemDelegations` function of `DelegationManager` is gas-heavy, there is a risk that the block gas limit may occur with the tokens that are at the end of the `terms` data. When the enforcer reverts because of the gas block limit, the `delegate` would be unable to move funds of the `delegator` in the delegation, so the `delegator` will have to create a new delegation for this exact token.

Examples

src/enforcers/MultiTokenPeriodEnforcer.sol:L157-L187

```
function getTermsInfo(
    bytes calldata _terms,
    address _token
)
    public
    pure
    returns (uint256 periodAmount_, uint256 periodDuration_, uint256 startDate_)
{
    uint256 termsLength_ = _terms.length;
    require(termsLength_ != 0 && termsLength_ % 116 == 0, "MultiTokenPeriodEnforcer:invalid-terms-length");

    // Iterate over the byte offset directly in increments of 116 bytes.
    for (uint256 offset_ = 0; offset_ < termsLength_;) {
        // Extract token address from the first 20 bytes.
        address token_ = address(bytes20(_terms[offset_:offset_ + 20]));
        if (token_ == _token) {
            // Get periodAmount from the next 32 bytes.
            periodAmount_ = uint256(bytes32(_terms[offset_ + 20:offset_ + 52]));
            // Get periodDuration from the subsequent 32 bytes.
            periodDuration_ = uint256(bytes32(_terms[offset_ + 52:offset_ + 84]));
            // Get startDate from the final 32 bytes.
            startDate_ = uint256(bytes32(_terms[offset_ + 84:offset_ + 116]));
            return (periodAmount_, periodDuration_, startDate_);
        }

        unchecked {
            offset_ += 116;
        }
    }
    revert("MultiTokenPeriodEnforcer:token-config-not-found");
}
```

Recommendation

We recommend optimizing the function by allowing an index parameter, enabling direct access to the relevant token. This will reduce unnecessary iteration and improve gas efficiency. The index variable can be passed as an `args` variable into the enforcer.

4.3 Missing Slippage Protection in Token Swap Execution Medium Acknowledged

Resolution

Acknowledged by the client with the following note:

> We do not support deflationary tokens. We are trusting the MetaSwap API to give proper swap data, including slippage.

Description

The `swapTokens` function performs a token swap via the `metaSwap` contract of Metamask and forwards the output tokens to a recipient. However, the function does not implement any slippage protection. Slippage protection is a critical safeguard in decentralized token swaps, ensuring that users do not receive an unreasonably low amount of output tokens due to price movement, poor liquidity, or malicious `delegate`. The fact that the data is signed by API signer helps to ensure that the calldata itself is not compromised, however, there is no validation that the tokens have been returned as expected. This check should be present in the `metaSwap` contract, but not adding this validation will make this contract fully dependent on the trust of Metamask and its 3rd parties. Additionally, if a deflationary token is swapped, the user might receive less than the minimum amount expected. This is because an additional transfer occurs: from the `metaSwap` to the `DelegationMetaSwapAdapter,` and only then to the user. Due to the deflation, this extra transfer may consume more tokens than the minimum amount specified in the `metaSwap` contract.

Examples

src/helpers/DelegationMetaSwapAdapter.sol:L280-L317

```
function swapTokens(
    string calldata _aggregatorId,
    IERC20 _tokenFrom,
    IERC20 _tokenTo,
    address _recipient,
    uint256 _amountFrom,
    uint256 _balanceFromBefore,
    bytes calldata _swapData
)
    external
    onlySelf
{
    uint256 tokenFromObtained_ = _getSelfBalance(_tokenFrom) - _balanceFromBefore;
    if (tokenFromObtained_ < _amountFrom) revert InsufficientTokens();

    if (tokenFromObtained_ > _amountFrom) {
        _sendTokens(_tokenFrom, tokenFromObtained_ - _amountFrom, _recipient);
    }

    uint256 balanceToBefore_ = _getSelfBalance(_tokenTo);

    uint256 value_ = 0;

    if (address(_tokenFrom) == address(0)) {
        value_ = _amountFrom;
    } else {
        uint256 allowance_ = _tokenFrom.allowance(address(this), address(metaSwap));
        if (allowance_ < _amountFrom) {
            _tokenFrom.safeIncreaseAllowance(address(metaSwap), type(uint256).max);
        }
    }

    metaSwap.swap{ value: value_ }(_aggregatorId, _tokenFrom, _amountFrom, _swapData);

    uint256 obtainedAmount_ = _getSelfBalance(_tokenTo) - balanceToBefore_;

    _sendTokens(_tokenTo, obtainedAmount_, _recipient);
}
```

src/helpers/DelegationMetaSwapAdapter.sol:L481-L499

```
// Excluding the function selector
bytes memory paramTerms_ = _apiData[4:];
(aggregatorId_, tokenFrom_, amountFrom_, swapData_) = abi.decode(paramTerms_, (string, IERC20, uint256, bytes));

// Note: Prepend address(0) to format the data correctly because of the Swaps API. See internal docs.
(
    , // address(0)
    IERC20 swapTokenFrom_,
    IERC20 swapTokenTo_,
    uint256 swapAmountFrom_,
    , // AmountTo
    , // Metadata
    uint256 feeAmount_,
    , // FeeWallet
    bool feeTo_
) = abi.decode(
    abi.encodePacked(abi.encode(address(0)), swapData_),
    (address, IERC20, IERC20, uint256, uint256, bytes, uint256, address, bool)
);
```

```
// Metaswap contract
    function swap(
        string calldata aggregatorId,
        IERC20 tokenFrom,
        uint256 amount,
        bytes calldata data
    ) external payable whenNotPaused nonReentrant {
        _swap(aggregatorId, tokenFrom, amount, data);
    }
```

Recommendation

We recommend adding a `minAmountOut` parameter to the function and validating the amount received after the swap. For example:

```
function swapTokens(
    string calldata _aggregatorId,
    IERC20 _tokenFrom,
    IERC20 _tokenTo,
    address _recipient,
    uint256 _amountFrom,
    uint256 _balanceFromBefore,
    uint256 _minAmountOut,
    bytes calldata _swapData
) external onlySelf {
    ...
    uint256 obtainedAmount_ = _getSelfBalance(_tokenTo) - balanceToBefore_;
    require(obtainedAmount_ >= _minAmountOut, "Slippage exceeded");
    ...
}
```

This ensures the swap meets user expectations and mitigates the risk of significant loss due to market fluctuations or malicious behavior.

4.4 Metamask Aggregator Signature for Swap API Data Can Be Re-Used Medium Acknowledged

Resolution

Acknowledged by the client with the following note:

> We just want to make sure the swaps apiData comes from our MetaSwap API. It is fine if it were to be reused.

Description

The `DelegationMetaSwapAdapter` contract uses swap data that is signed by the Metamask swap aggregator to facilitates the swaps for delegators and their delegates. To do so, the `_validateSignature()` function is called during the execution of `swapByDelegation()`:

src/helpers/DelegationMetaSwapAdapter.sol:L215-L222

```
function swapByDelegation(
    SignatureData calldata _signatureData,
    Delegation[] memory _delegations,
    bool _useTokenWhitelist
)
    external
{
    _validateSignature(_signatureData);
```

However, while the check verifies the signatures against the signed data and its expiration status, it does not check whether or not the signature has already been used before:

src/helpers/DelegationMetaSwapAdapter.sol:L522-L534

```
/**
 * @dev Validates the expiration and signature of the provided apiData.
 * @param _signatureData Contains the apiData, the expiration and signature.
 */
function _validateSignature(SignatureData memory _signatureData) private view {
    if (block.timestamp > _signatureData.expiration) revert SignatureExpired();

    bytes32 messageHash_ = keccak256(abi.encodePacked(_signatureData.apiData, _signatureData.expiration));
    bytes32 ethSignedMessageHash_ = MessageHashUtils.toEthSignedMessageHash(messageHash_);

    address recoveredSigner_ = ECDSA.recover(ethSignedMessageHash_, _signatureData.signature);
    if (recoveredSigner_ != swapApiSigner) revert InvalidApiSignature();
}
```

As a result, the same signature with the same signed swap data, including prices, can be re-used, even in the same block. That said, the price data shouldn't be stale as there is a time expiration check. Still, it is best practice not to re-use signatures and flag them as used instead, such as via a stored nonce.

Recommendation

Mark Metamask aggregator's API swap data signatures as used after validating them.

4.5 Avoid `abi.encodePacked()` With Dynamic Types for Hashing Minor ✓ Fixed

Resolution

Fixed in commit 5befce72aceb8851895e6cdcbae63d0ec47ef67a by implementing `abi.encode()` instead of `abi.encodePacked()`.

Description

The contract uses `abi.encodePacked()` with dynamic types in hashing operations. This pattern is discouraged because `abi.encodePacked()` does not include length information for dynamic types, making it vulnerable to hash collisions. As a result, different sets of input data may produce identical hashes, potentially leading to critical security issues. This is especially problematic when passing the result to `keccak256()`, as it can undermine the uniqueness and integrity of signature verifications and other sensitive logic. The `_validateSignature` function is susceptible to a hash collision, during the encoding of `_signatureData.apiData` and `_signatureData.expiration` variables; however, due to a lack information, it's impossible to create a real scenario. Still, it is recommended to change the type of encoding to be safe.

Examples

src/helpers/DelegationMetaSwapAdapter.sol:L526-L534

```
function _validateSignature(SignatureData memory _signatureData) private view {
    if (block.timestamp > _signatureData.expiration) revert SignatureExpired();

    bytes32 messageHash_ = keccak256(abi.encodePacked(_signatureData.apiData, _signatureData.expiration));
    bytes32 ethSignedMessageHash_ = MessageHashUtils.toEthSignedMessageHash(messageHash_);

    address recoveredSigner_ = ECDSA.recover(ethSignedMessageHash_, _signatureData.signature);
    if (recoveredSigner_ != swapApiSigner) revert InvalidApiSignature();
}
```

Recommendation

We recommend replacing `abi.encodePacked()` with `abi.encode()` when hashing multiple dynamic types to avoid potential collisions and ensure consistency.

4.6 Missing Input Validation in `constructor` and Setter Function Minor ✓ Fixed

Resolution

Fixed in commit 6912e732e2ed65699152c6bfdb46a0ed433f1263 by adding 0-address checks.

Description

The `constructor` sets critical contract dependencies such as `swapApiSigner`, `delegationManager`, `metaSwap`, and `argsEqualityCheckEnforcer` without validating the provided addresses. This could result in the contract being initialized with zero addresses or incorrect contracts, leading to broken functionality or security vulnerabilities.

Similarly, the `setSwapApiSigner` function allows setting a new signer address without validating that it is non-zero.

Examples

src/helpers/DelegationMetaSwapAdapter.sol:L178-L190

```
constructor(
    address _owner,
    address _swapApiSigner,
    IDelegationManager _delegationManager,
    IMetaSwap _metaSwap,
    address _argsEqualityCheckEnforcer
)
    Ownable(_owner)
{
    swapApiSigner = _swapApiSigner;
    delegationManager = _delegationManager;
    metaSwap = _metaSwap;
    argsEqualityCheckEnforcer = _argsEqualityCheckEnforcer;
```

src/helpers/DelegationMetaSwapAdapter.sol:L323-L326

```
function setSwapApiSigner(address _newSigner) external onlyOwner {
    swapApiSigner = _newSigner;
    emit SwapApiSignerUpdated(_newSigner);
}
```

Recommendation

We recommend adding input validation checks to ensure none of the critical addresses passed to the constructor or `setSwapApiSigner` are the zero address.

4.7 Missing Deadline Enforcement in Periodic Allowance Logic Minor Acknowledged

Resolution

Acknowledged by the client team with the following note:

> This is intentional. Delegators can add the timestamp enforcer if they want an expiration.

Description

The `PeriodicAllowance` struct defines parameters for enforcing periodic transfer limits, including `periodAmount`, `periodDuration`, `startDate`, and tracking variables like `lastTransferPeriod` and `transferredInCurrentPeriod`. However, there is no field for a `deadline` or end time that would allow the system to enforce when a periodic allowance should expire. As a result, once a periodic allowance is set, it can remain valid indefinitely unless explicitly revoked or replaced, which could cause inconveniences for the users.

This omission can introduce unwanted behavior or risk, especially in cases where the allowance should only be valid for a specific timeframe, such as temporary access or time-limited delegations.

Examples

src/enforcers/MultiTokenPeriodEnforcer.sol:L41-L47

```
struct PeriodicAllowance {
    uint256 periodAmount; // Maximum transferable amount per period.
    uint256 periodDuration; // Duration of each period in seconds.
    uint256 startDate; // Timestamp when the first period begins.
    uint256 lastTransferPeriod; // The period index in which the last transfer was made.
    uint256 transferredInCurrentPeriod; // Cumulative amount transferred in the current period.
}
```

Recommendation

We recommend adding a `deadline` field to the `PeriodicAllowance` struct and enforcing it in the allowance logic as an option. This would enable more flexible and secure time-based controls for delegation and token transfer rules, ensuring allowances cannot be abused or left open-ended beyond their intended scope when needed.

4.8 Replace Revert Strings With Custom Errors for Gas Optimization Acknowledged

Resolution

Acknowledged by the client team with the following note:

> This is intentional all enforcers are using string errors for better readability.

Description

The `MultiTokenPeriodEnforcer` contract currently uses standard revert strings to signal errors. While informative, revert strings consume more gas due to memory allocation and storage requirements. Since Solidity version 0.8.4, custom errors have been introduced as a more gas-efficient alternative, saving approximately 50 gas per occurrence and reducing deployment costs. Additionally, custom errors improve consistency and can be used across contracts, interfaces, and libraries. Also, custom errors are already used in other parts of the codebase.

Examples

src/enforcers/MultiTokenPeriodEnforcer.sol:L166

```
require(termsLength_ != 0 && termsLength_ % 116 == 0, "MultiTokenPeriodEnforcer:invalid-terms-length");
```

src/enforcers/MultiTokenPeriodEnforcer.sol:L209

```
require(termsLength_ % 116 == 0 && termsLength_ != 0, "MultiTokenPeriodEnforcer:invalid-terms-length");
```

src/enforcers/MultiTokenPeriodEnforcer.sol:L260-L261

```
require(value_ == 0, "MultiTokenPeriodEnforcer:invalid-value-in-erc20-transfer");
require(bytes4(callData_[0:4]) == IERC20.transfer.selector, "MultiTokenPeriodEnforcer:invalid-method");
```

src/enforcers/MultiTokenPeriodEnforcer.sol:L266

```
require(value_ > 0, "MultiTokenPeriodEnforcer:invalid-zero-value-in-native-transfer");
```

src/enforcers/MultiTokenPeriodEnforcer.sol:L282-L285

```
require(startDate_ > 0, "MultiTokenPeriodEnforcer:invalid-zero-start-date");
require(periodAmount_ > 0, "MultiTokenPeriodEnforcer:invalid-zero-period-amount");
require(periodDuration_ > 0, "MultiTokenPeriodEnforcer:invalid-zero-period-duration");
require(block.timestamp >= startDate_, "MultiTokenPeriodEnforcer:transfer-not-started");
```

src/enforcers/MultiTokenPeriodEnforcer.sol:L294

```
require(transferAmount_ <= availableAmount_, "MultiTokenPeriodEnforcer:transfer-amount-exceeded");
```

src/enforcers/MultiTokenPeriodEnforcer.sol:L271

```
revert("MultiTokenPeriodEnforcer:invalid-call-data-length");
```

Recommendation

We recommend replacing all revert strings with custom error definitions. Adopting custom errors will reduce gas usage, improve maintainability, and standardize error reporting across the contract.

4.9 Replace `abi.encodeWithSignature` and `abi.encodeWithSelector` With `abi.encodeCall` for Type and Typo Safety ✓ Fixed

Resolution

Fixed in a553f4c58513105c454448bcd4a180a894593809 by using `abi.encodeCall()` instead of `abi.encodeWithSelector()`.

Description

The contract uses `abi.encodeWithSelector` to construct low-level calls. While functional, this approach is error-prone as it does not offer compile-time checks for function names or parameter types. Typos in the function selector or mismatched parameter types can go unnoticed until runtime, potentially causing silent failures or unexpected behavior.

The more modern and type-safe alternative is `abi.encodeCall`, introduced in Solidity 0.8.12. This method enforces both function name correctness and argument type safety at compile time.

Examples

```
// DelegationMetaSwapAdapter.sol

bytes memory encodedSwap_ = abi.encodeWithSelector(
bytes memory encodedTransfer_ = abi.encodeWithSelector(IERC20.transfer.selector, address(this), amountFrom_);
```

Recommendation

We recommend replacing `abi.encodeWithSelector` with `abi.encodeCall` for safer and more maintainable code. For example:

```
abi.encodeCall(IERC20.transfer, (address(this), amountFrom_));
```

4.10 `public` Function Can Be Declared `external` ✓ Fixed

Resolution

Fixed in c4cfef05b1a256d476b7c25ed49d1b95d6d5a801 by changing function visibilities from `public` to `external`.

Description

The function `getAllTermsInfo` in the `MultiTokenPeriodEnforcer` contract is declared as `public` but is not called internally within the contract. Declaring this function as `external` would save gas and more accurately reflect its intended usage, as it is meant to be invoked externally.

Examples

src/enforcers/MultiTokenPeriodEnforcer.sol:L189-L207

```
/**
 * @notice Decodes all configurations contained in _terms.
 * @dev Expects _terms length to be a multiple of 116.
 * @param _terms A concatenation of 116-byte configurations.
 * @return tokens_ An array of token addresses.
 * @return periodAmounts_ An array of period amounts.
 * @return periodDurations_ An array of period durations (in seconds).
 * @return startDates_ An array of start dates for the first period.
 */
function getAllTermsInfo(bytes calldata _terms)
    public
    pure
    returns (
        address[] memory tokens_,
        uint256[] memory periodAmounts_,
        uint256[] memory periodDurations_,
        uint256[] memory startDates_
    )
{
```

Recommendation

We recommend changing the visibility of the following functions from `public` to `external`.

4.11 Typos in Codebase ✓ Fixed

Resolution
Fixed in 8cda2fec17c9bbc34454da63c29ab235a61facad by addressing the typos.

Description

There are typographical errors in the names of custom errors `TokenFromMismath` and `AmountFromMismath`. The word “Mismath” appears to be a misspelling of “Mismatch.”

Examples

src/helpers/DelegationMetaSwapAdapter.sol:L134-L138

```
/// @dev Error when the tokenFrom in the api data and swap data do not match.
error TokenFromMismath();

/// @dev Error when the amountFrom in the api data and swap data do not match.
error AmountFromMismath();
```

Recommendation

We recommend renaming the custom errors to correct the spelling and to keep the codebase clean:

```
error TokenFromMismatch();
error AmountFromMismatch();
```

4.12 Inconsistent Transfer Validation Across Call Types Acknowledged

Resolution
Acknowledged by the client team with the following note: For the individual period enforcers delegators can combine them with other caveats to enforcer the calldata or value.

Description

In the `_validateAndConsumeTransfer` logic, the contract includes specific checks for ERC-20 and native transfers. For ERC-20 transfers, it validates that `value_` is zero and the function selector matches `IERC20.transfer.selector`. For native transfers, it enforces that `value_ > 0`. These validations help prevent malformed or unexpected calldata execution.

However, similar checks are not consistently applied in other parts of the codebase, which is currently out-of-scope, that handle token or native transfers. This inconsistency may lead to uneven enforcement of validation rules, allowing potentially invalid calls to succeed elsewhere in the system, increasing the surface area for bugs or exploits.

Examples

src/enforcers/MultiTokenPeriodEnforcer.sol:L258-L272

```
if (callData_.length == 68) {
    // ERC20 transfer.
    require(value_ == 0, "MultiTokenPeriodEnforcer:invalid-value-in-erc20-transfer");
    require(bytes4(callData_[0:4]) == IERC20.transfer.selector, "MultiTokenPeriodEnforcer:invalid-method");
    token_ = target_;
    transferAmount_ = uint256(bytes32(callData_[36:68]));
} else if (callData_.length == 0) {
    // Native transfer.
    require(value_ > 0, "MultiTokenPeriodEnforcer:invalid-zero-value-in-native-transfer");
    token_ = address(0);
    transferAmount_ = value_;
} else {
    // If callData length is neither 68 nor 0, revert.
    revert("MultiTokenPeriodEnforcer:invalid-call-data-length");
}
```

src/enforcers/NativeTokenPeriodTransferEnforcer.sol:L161-L212

```
function _validateAndConsumeTransfer(
    bytes calldata _terms,
    bytes calldata _executionCallData,
    bytes32 _delegationHash,
    address _redeemer
)
    private
{
    (, uint256 value_,) = _executionCallData.decodeSingle();

    (uint256 periodAmount_, uint256 periodDuration_, uint256 startDate_) = getTermsInfo(_terms);

    PeriodicAllowance storage allowance_ = periodicAllowances[msg.sender][_delegationHash];

    // Initialize the allowance on first use.
    if (allowance_.startDate == 0) {
        // Validate terms.
        require(startDate_ > 0, "NativeTokenPeriodTransferEnforcer:invalid-zero-start-date");
        require(periodAmount_ > 0, "NativeTokenPeriodTransferEnforcer:invalid-zero-period-amount");
        require(periodDuration_ > 0, "NativeTokenPeriodTransferEnforcer:invalid-zero-period-duration");

        // Ensure the transfer period has started.
        require(block.timestamp >= startDate_, "NativeTokenPeriodTransferEnforcer:transfer-not-started");

        allowance_.periodAmount = periodAmount_;
        allowance_.periodDuration = periodDuration_;
        allowance_.startDate = startDate_;
    }

    // Calculate available ETH using the current allowance state.
    (uint256 available_, bool isNewPeriod_, uint256 currentPeriod_) = _getAvailableAmount(allowance_);

    require(value_ <= available_, "NativeTokenPeriodTransferEnforcer:transfer-amount-exceeded");

    // If a new period has started, update state before processing the transfer.
    if (isNewPeriod_) {
        allowance_.lastTransferPeriod = currentPeriod_;
        allowance_.transferredInCurrentPeriod = 0;
    }
    allowance_.transferredInCurrentPeriod += value_;

    emit TransferredInPeriod(
        msg.sender,
        _redeemer,
        _delegationHash,
        periodAmount_,
        periodDuration_,
        allowance_.startDate,
        allowance_.transferredInCurrentPeriod,
        block.timestamp
    );
}
```

src/enforcers/ERC20PeriodTransferEnforcer.sol:L173-L231

```
function _validateAndConsumeTransfer(
    bytes calldata _terms,
    bytes calldata _executionCallData,
    bytes32 _delegationHash,
    address _redeemer
)
    private
{
    (address target_,, bytes calldata callData_) = _executionCallData.decodeSingle();

    require(callData_.length == 68, "ERC20PeriodTransferEnforcer:invalid-execution-length");

    (address token_, uint256 periodAmount_, uint256 periodDuration_, uint256 startDate_) = getTermsInfo(_terms);

    require(token_ == target_, "ERC20PeriodTransferEnforcer:invalid-contract");
    require(bytes4(callData_[0:4]) == IERC20.transfer.selector, "ERC20PeriodTransferEnforcer:invalid-method");

    PeriodicAllowance storage allowance_ = periodicAllowances[msg.sender][_delegationHash];

    // Initialize the allowance on first use.
    if (allowance_.startDate == 0) {
        require(startDate_ > 0, "ERC20PeriodTransferEnforcer:invalid-zero-start-date");
        require(periodAmount_ > 0, "ERC20PeriodTransferEnforcer:invalid-zero-period-amount");
        require(periodDuration_ > 0, "ERC20PeriodTransferEnforcer:invalid-zero-period-duration");

        // Ensure the transfer period has started.
        require(block.timestamp >= startDate_, "ERC20PeriodTransferEnforcer:transfer-not-started");

        allowance_.periodAmount = periodAmount_;
        allowance_.periodDuration = periodDuration_;
        allowance_.startDate = startDate_;
    }

    // Calculate available tokens using the current allowance state.
    (uint256 available_, bool isNewPeriod_, uint256 currentPeriod_) = _getAvailableAmount(allowance_);

    uint256 transferAmount_ = uint256(bytes32(callData_[36:68]));
    require(transferAmount_ <= available_, "ERC20PeriodTransferEnforcer:transfer-amount-exceeded");

    // If a new period has started, reset transferred amount before continuing.
    if (isNewPeriod_) {
        allowance_.lastTransferPeriod = currentPeriod_;
        allowance_.transferredInCurrentPeriod = 0;
    }

    allowance_.transferredInCurrentPeriod += transferAmount_;

    emit TransferredInPeriod(
        msg.sender,
        _redeemer,
        _delegationHash,
        token_,
        periodAmount_,
        periodDuration_,
        startDate_,
        allowance_.transferredInCurrentPeriod,
        block.timestamp
    );
}
```

Recommendation

We recommend reviewing all transfer-related code paths and applying consistent validation logic across the codebase. Reuse the same safety checks (e.g., for `value_`, function selectors, and expected calldata length) to ensure predictable and safe execution regardless of where the transfer originates. This promotes uniformity and minimizes risk.

4.13 Re-Entrancy Risk in `swapTokens()` Acknowledged

Resolution

Acknowledged by the client team to have little relevant risk with the following note:

> There could be a chance for reentrancy but it would be from the root delegator, so we're not sure what the risk is there.

Description

The `DelegationMetaSwapAdapter` contract facilitates swaps for delegators and delegates. As part of that, tokens are deposited first to the adapter before a swap is executed. As a result, there is a balance check to ensure that the tokens received in the contract are sufficient:

src/helpers/DelegationMetaSwapAdapter.sol:L280-L297

```
function swapTokens(
    string calldata _aggregatorId,
    IERC20 _tokenFrom,
    IERC20 _tokenTo,
    address _recipient,
    uint256 _amountFrom,
    uint256 _balanceFromBefore,
    bytes calldata _swapData
)
    external
    onlySelf
{
    uint256 tokenFromObtained_ = _getSelfBalance(_tokenFrom) - _balanceFromBefore;
    if (tokenFromObtained_ < _amountFrom) revert InsufficientTokens();

    if (tokenFromObtained_ > _amountFrom) {
        _sendTokens(_tokenFrom, tokenFromObtained_ - _amountFrom, _recipient);
    }
```

In fact, if there are more tokens received than needed, i.e. `tokenFromObtained_ > _amountFrom` , they are sent back to the recipient via `_sendTokens()`. However, if the intended token is the native token of the chain, for example ETH, then the `_sendTokens()` performs a `.call()` to the recipient:

src/helpers/DelegationMetaSwapAdapter.sol:L421-L431

```
function _sendTokens(IERC20 _token, uint256 _amount, address _recipient) private {
    if (address(_token) == address(0)) {
        (bool success_,) = _recipient.call{ value: _amount }("");

        if (!success_) revert FailedNativeTokenTransfer(_recipient);
    } else {
        IERC20(_token).safeTransfer(_recipient, _amount);
    }

    emit SentTokens(_token, _recipient, _amount);
}
```

In other words, an external call is made, which opens up re-entrancy risks. In fact, this recipient could re-initiate the whole flow on the same contracts, and perform another swap utilizing `DelegationMetaSwapAdapter`, which would have tokens from the previous swap still there before it is finished.

While no major attack vector was identified during the audit, unless there is a specific use case for allowing re-entrancy to the contract mid-swap, we suggest placing a re-entrancy guard on `swapTokens()`.

Recommendation

Consider placing a re-entrancy guard on `swapTokens()`.

Appendix 1 - Files in Scope

This audit covered the following files:

File	SHA-1 hash
src/enforcers/MultiTokenPeriodEnforcer.sol	`eb09b0e2ec67d2456cbedbaca4b67ace81fb010d`
src/helpers/DelegationMetaSwapAdapter.sol	`184f88ee43542e0351c17be2a8f42946db2da944`

Appendix 2 - Disclosure

Consensys Diligence ("CD") typically receives compensation from one or more clients (the "Clients") for performing the analysis contained in these reports (the "Reports"). The Reports may be distributed through other means, including via Consensys publications and other distributions.

The Reports are not an endorsement or indictment of any particular project or team, and the Reports do not guarantee the security of any particular project. This Report does not consider, and should not be interpreted as considering or having any bearing on, the potential economics of a token, token sale or any other product, service or other asset. Cryptographic tokens are emergent technologies and carry with them high levels of technical risk and uncertainty. No Report provides any warranty or representation to any third party in any respect, including regarding the bug-free nature of code, the business model or proprietors of any such business model, and the legal compliance of any such business. No third party should rely on the Reports in any way, including for the purpose of making any decisions to buy or sell any token, product, service or other asset. Specifically, for the avoidance of doubt, this Report does not constitute investment advice, is not intended to be relied upon as investment advice, is not an endorsement of this project or team, and it is not a guarantee as to the absolute security of the project. CD owes no duty to any third party by virtue of publishing these Reports.

A.2.1 Purpose of Reports

The Reports and the analysis described therein are created solely for Clients and published with their consent. The scope of our review is limited to a review of code and only the code we note as being within the scope of our review within this report. Any Solidity code itself presents unique and unquantifiable risks as the Solidity language itself remains under development and is subject to unknown risks and flaws. The review does not extend to the compiler layer, or any other areas beyond specified code that could present security risks. Cryptographic tokens are emergent technologies and carry with them high levels of technical risk and uncertainty. In some instances, we may perform penetration testing or infrastructure assessments depending on the scope of the particular engagement.

CD makes the Reports available to parties other than the Clients (i.e., "third parties") on its website. CD hopes that by making these analyses publicly available, it can help the blockchain ecosystem develop technical best practices in this rapidly evolving area of innovation.

A.2.2 Links to Other Web Sites from This Web Site

You may, through hypertext or other computer links, gain access to web sites operated by persons other than Consensys and CD. Such hyperlinks are provided for your reference and convenience only, and are the exclusive responsibility of such web sites' owners. You agree that Consensys and CD are not responsible for the content or operation of such Web sites, and that Consensys and CD shall have no liability to you or any other person or entity for the use of third party Web sites. Except as described below, a hyperlink from this web Site to another web site does not imply or mean that Consensys and CD endorses the content on that Web site or the operator or operations of that site. You are solely responsible for determining the extent to which you may use any content at any other web sites to which you link from the Reports. Consensys and CD assumes no responsibility for the use of third-party software on the Web Site and shall have no liability whatsoever to any person or entity for the accuracy or completeness of any outcome generated by such software.

A.2.3 Timeliness of Content

The content contained in the Reports is current as of the date appearing on the Report and is subject to change without notice unless indicated otherwise, by Consensys and CD.