Delegation

Date	February 2025

1 Executive Summary

This report presents the results of our engagement with **MetaMask** to review **delegation framework**.

The review was conducted over two weeks, from **February 17, 2025** to **March 6, 2025**, by **Rai Yang** and **Sergii Kravchenko**. A total of 30 person-days were spent.

The review is performed on the changes in the codebase that were made since the previous audit commit. The main changes are:

- Enabling EOA to set the code to a delegator account under EIP 7702.
- Adding three new enforcers contracts
- Adding an intermediary contract to performs delegation-based swaps on MetaSwap , particularly for a use case where a MetaMask vault delegates to a sub-vault to perform specific swap actions.

While enforcer contracts provide restrictions on delegations, we found a critical vulnerablity in the intermediary contract that allows attacker to front-run the swap transaction and manipulate swap data, leading to various financial losses to the delegator and disrupting the original swap transaction.

Mitiagtions. As the part of the fixes, the team introduced new smart contracts: `ExactCalldataBatchEnforcer` and `ExactCalldataEnforcer` that are used in combination with the `NativeTokenStreamingEnforcer`. The details of how every issue was addressed can be found inside every individual issue.

2 Scope

Our review focused on the commit hash d522a38b0b0f1c27d896790262302a52c3720e06. 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 System Overview

This section describes the changes that were examined in the scope of this review.

3.1 EIP-7702 support

Overview

The upcoming Ethereum upgrade will introduce **EIP-7702**, a proposal that enables externally owned accounts (EOAs) to function as smart contracts. This is achieved by setting the account's code to the following value: `0xef0100 || address`, where `address` represents the smart contract to which code execution will be delegated.

To adapt to this change, the **delegation framework** has been updated to support EIP-7702-compatible addresses.

Previous Smart Account Architecture

Traditionally, **smart accounts** within the delegation framework operated as **proxy contracts** that pointed to implementations inheriting from the `DeleGatorCore` contract. This contract encapsulated the core logic required for delegatable smart accounts. The **UUPSUpgradeable** proxy pattern was utilized to facilitate upgrades.

EIP-7702 Adaptation

With the introduction of **EIP-7702**, an alternative contract structure was introduced:

- `EIP7702DeleGatorCore` **(Abstract Contract):** Contains the fundamental delegation logic.
- `EIP7702StatelessDeleGator` **(Implementation):** Inherits from `EIP7702DeleGatorCore` and serves as the implementation for EIP-7702-based accounts.

While `EIP7702DeleGatorCore` closely resembles `DeleGatorCore`, the key distinction lies in its proxy mechanism. Instead of utilizing **UUPSUpgradeable proxies**, EIP-7702 smart accounts are transformed **directly from EOAs**. These accounts must explicitly create a **delegation designation**: `0xef0100 || address`, where `address` refers to `EIP7702StatelessDeleGator`. This allows EOA to be a delegator and ensures proper delegation and execution within the new framework.

3.2 New Caveat Enforcers

There are three new caveat enforcers introduced:

- `ERC20StreamingEnforcer`
- `NativeTokenStreamingEnforcer`
- `SpecificActionERC20TransferBatchEnforcer`

The first two enforcers share similar logic, with the key difference being that `ERC20StreamingEnforcer` operates with **ERC-20 tokens**, while `NativeTokenStreamingEnforcer` handles **native tokens** (ETH on Ethereum mainnet) directly.

`ERC20StreamingEnforcer` and `NativeTokenStreamingEnforcer`

These enforcers are designed to facilitate **delegations for streaming payments**. Within this delegation model, tokens are released over time, allowing the delegate to transfer unlocked tokens to themselves.

The delegation includes the following key parameters:

- `initialAmount` – The starting amount available for streaming after `startTime`.
- `maxAmount` – The maximum amount that can be streamed.
- `amountPerSecond` – The rate at which tokens are released over time.
- `startTime` – The timestamp when the streaming begins.

While these enforcers can be combined with others, there is **no strict requirement** imposed by the development team to do so.

`SpecificActionERC20TransferBatchEnforcer`

The `SpecificActionERC20TransferBatchEnforcer` is designed for a highly specific use case. It enforces a delegation that allows exactly **two execution calls** in a predefined batch:

1. **First Call:**
   - Must have a **zero value**.
   - The **address** and **calldata** must exactly match the values specified in the delegation terms.
2. **Second Call:**
   - Must be an **ERC-20 token transfer**.
   - The **token address**, **amount**, and **recipient** must also be explicitly defined in the delegation terms.

3.3 `DelegationMetaSwapAdapter`

The `DelegationMetaSwapAdapter` contract is designed to facilitate token swaps through the `metaSwap` contract. To operate, it requires a **delegator account** to create a **delegation** that authorizes `DelegationMetaSwapAdapter` to transfer **ERC-20 or native tokens** to itself.

Once authorized and redeemed, `DelegationMetaSwapAdapter` will:

1. **Transfer the delegated tokens to itself**.
2. **Swap the tokens using** `metaSwap`.
3. **Send the swapped funds back to the delegator**.

Anyone can trigger this process by calling `DelegationMetaSwapAdapter.swapByDelegation`, provided they have a **valid signed delegation** permitting the contract to execute the swap.

This contract is implemented as a **singleton**, maintaining a **list of approved tokens and aggregators** curated by the **MetaMask team**. Additionally, the contract's `owner` has the ability to **withdraw all funds** from the contract.

While `DelegationMetaSwapAdapter` is designed to support multiple use cases, there is a **specific primary use case** proposed by the team, which is detailed in the next section.

4 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.

The `DelegationMetaSwapAdapter` contract is intended to function as a singleton, serving multiple purposes for various actors. However, its design was originally tailored for a highly specific use case that is described further:

4.1 Actors

The relevant actors are listed below with their respective abilities:

- Vault: a secure Multi-Sig account (2/3) that controls user's asset and gives delegation to sub vault perform specific actions.
  - Target enforcer (token contract).
  - Method enforcer (`transfer`).
  - Calldata enforcer (to make the to `address == DelegationMetaSwapAdapter` address).
  - Redeemer enforcer (enforcing `DelegationMetaSwapAdapter` only can redeem).
- Sub-vault: a Multi-Sig account (1/3) performs action on behave of the main vault by delegation.
  - Calldata enforcer (enforcing the specific amount of tokens allowed to transfer out).
- Immediary contract: a contract receiving delegation from the Sub-vault and perform specific swap transaction.

4.2 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:

- Metamask team is trusted to maintain the non-malicious list of approved tokens and aggregators in the `DelegationMetaSwapAdapter`.

5 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.

5.1 `NativeTokenStreamingEnforcer` Allows Unintended Calls Leading to Loss of Funds Critical

Partially Addressed

Resolution

The team decided to address this by recommending combining this enforcer with the `ExactCalldataEnforcer` and setting it to revert with anything other than empty `calldata`. Additionally, the `AllowedTargetsEnforcer` can be used to restrict the target if needed.

Description

The `NativeTokenStreamingEnforcer` contract only checks if the `value` of the call is approved to be transferred but allows any `target` and `calldata` of the call and doesn't check it:

../code-1/src/enforcers/NativeTokenStreamingEnforcer.sol:L140-L151

```
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);
```

Due to that flexibility, many different attacks are possible; one of them would be to call `token.transfer` for any token, with any amount and recipient. That would drain all the tokens from the contract.

While `target` and `calldata` could also be checked by other enforcers, that's not intended by default and is unobvious for users who construct delegations.

The issue is also relevant to the `NativeTokenTransferAmountEnforcer` contract.

Recommendation

The attack vectors are very broad, and predicting all possible consequences is problematic. We recommend adding two restrictions:

- Ensure the `value` is above zero, preventing the attacker from calling non-payable functions.
- Ensure the call data is empty so no extra functions are called during value transfer.

That still leaves some potential vectors for calling undesirable fallback functions or transferring funds to toxic accounts, but they are less severe. It's essential to mention risks to the users when constructing delegations and enforcers so they consider whitelisting targets in a separate enforcer.

5.2 Front Running Function swapByDelegation Leading to Financial Loss of the Delegator Critical

Partially Addressed

Resolution

The issue is partially fixed by only allowing leaf delegator to call `swapByDelegation` and Metamask team responded that `apiData` will be signed and verified in the function to prevent it's being tampered

Description

In the `DelegationMetaSwapAdapter` contract, anyone possessing a valid signed delegation can invoke the `swapByDelegation` function to swap tokens on behalf of the delegator. However, this introduces a front-running risk where attacker can front run the original transaction `swapByDelegation` modifying the `_apiData` parameters such as:

- Swap aggregator: Selecting a less efficient aggregator, leading to financial loss.
- Swap data: Modifying execution parameters to exploit slippage.
  - Altering slippage tolerance: The delegator may set a reasonable slippage tolerance, a front-runner can tamper with swap data to increase slippage tolerance (e.g., 10% or more) and executes swaps at much worse rates, causing significant financial losses or price differences.

- Sandwich attacks: A front-runner detects an upcoming `swapByDelegation` transaction and places a large buy order before it to inflate the token price, when `swapByDelegation` executes at the now-inflated price, the delegator pays more than necessary, the attacker then immediately sells after execution, profiting while the delegator suffers losses

Consequently the original `swapByDelegation` transaction would revert, disrupting original swaps.

Although enforcers will be added to restrict some swap parameters and there is an allowlist for allowed aggregator and tokens, the signed delegation still can be manipulated in many ways, particularly swap data as they are NOT included any delegation that can be restricted by enforcers, as well as replay attack. Furthermore if any relevant enforcer is missing, the attacker can cause much more financial damages to the delegator.

Examples

../code-1/src/helpers/DelegationMetaSwapAdapter.sol:L140-L182

```
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_);

    // 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_
    );

    bytes[] memory permissionContexts_ = new bytes[](2);
    permissionContexts_[0] = abi.encode(_delegations);
    permissionContexts_[1] = abi.encode(new Delegation[](0));

    ModeCode[] memory encodedModes_ = new ModeCode[](2);
    encodedModes_[0] = ModeLib.encodeSimpleSingle();
    encodedModes_[1] = ModeLib.encodeSimpleSingle();

    bytes[] memory executionCallDatas_ = new bytes[](2);

    if (address(tokenFrom_) == address(0)) {
        executionCallDatas_[0] = ExecutionLib.encodeSingle(address(this), amountFrom_, hex"");
    } else {
        bytes memory encodedTransfer_ = abi.encodeWithSelector(IERC20.transfer.selector, address(this), amountFrom_);
        executionCallDatas_[0] = ExecutionLib.encodeSingle(address(tokenFrom_), 0, encodedTransfer_);
    }
    executionCallDatas_[1] = ExecutionLib.encodeSingle(address(this), 0, encodedSwap_);

    delegationManager.redeemDelegations(permissionContexts_, encodedModes_, executionCallDatas_);
}
```

Recommendation

1. Add a validation for the caller of function `swapByDelegation` to be the delegator of the leaf delegate(`DelegationMetaSwapAdapter` contract)
2. Ensure swap data(e.g. `slippage`, `minAmountOut`) are signed into the `_apiData`'s signature to prevent it being tampered
3. Add enforcers to prevent replay attack and ensure swap transactions expire within a short time (e.g nonce enforcer, timestamp enforcer)

Appendix 1 - Files in Scope

This audit covered the following files:

File	SHA-1 hash
src/EIP7702/EIP7702DeleGatorCore.sol	`013cc766c0728fad4ebed720e5efa8e52b4e0c67`
src/EIP7702/EIP7702StatelessDeleGator.sol	`19d4c46d9e6a5f768e988475dea9d742c1f6358b`
src/enforcers/ERC20StreamingEnforcer.sol	`6cde79c139090aba76ec2b78c7a1c26728db33d0`
src/enforcers/NativeTokenStreamingEnforcer.sol	`ffe154f6430c518365ac132080cbbab65099cc20`
src/enforcers/SpecificActionERC20TransferBatchEnforcer.sol	`4fd59a7e5d3fd3c0d0d91aebaee8073e90c3bab3`
src/helpers/DelegationMetaSwapAdapter.sol	`7f0f563287f0af0d161d1a8ffd4a80fd5a6709c5`

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.