MetaMask DeleGator

Date	June 2024
Auditors	Heiko Fisch, Arturo Roura

1 Executive Summary

This report presents the results of our engagement with the **MetaMask DeleGator** team to review the smart contracts of their **DeleGator** codebase.

The review was conducted in **May and June, 2024**, by **Heiko Fisch** and **Arturo Roura**.

Very briefly, the DeleGator system allows to delegate control over a smart contract account to other parties. While this control can be unrestricted, in the vast majority of cases, the SCA owner will want to include caveats in the delegation that restrict what actions the delegate can perform on the account. This is achieved via enforcer contracts that have access to the action and contextual information regarding the delegation. While an enforcer's primary task is to decide whether to allow or reject the redemption of a delegation with this particular action, they are much more versatile and can contain arbitrary state-changing code, allowing for a highly flexible delegation system. Finally, a delegate can re-delegate their rights on the root SCA to other parties; new caveats can be added in this process and are enforced together with the original ones. This allows for entire delegation chains, where each delegator along the chain can add their own restrictions.

The premise of the system – giving someone else control over one's account – in combination with the high flexibility naturally creates a high-risk situation for assets held in and privileges associated with the SCA. We strongly recommend to exercise caution and limit exposure. Since caveats are responsible for the restrictions that accompany a delegation, getting them right is crucial, so delegators in general and the root delegator in particular must pay utmost attention to the caveats. This is particularly true since enforcers have a low-level interface and quite often a very technical function/specification, while delegators have high-level intentions for their restrictions. We see wrong enforcer usage – and insufficient delegation restriction in general – as one of the main security risks.

Please note that we have not conducted a review of the fixes proposed by the client for the identified issues. While the client has acknowledged some issues and has proposed fixes for others, these fixes have not been reviewed by us.

2 Scope

Our review focused on the commit hash ee9f363e1128c7ef214f36e08dc0ce0b1069ef26. Most of the contracts were in scope, with the exception of an unfinished enforcer; the detailed list of files together with their SHA-1 hashes can be found in the Appendix. Daimo's P256 verifier has been audited before by Veridise and was included in this review only on a best-effort basis.

3 System Overview

3.1 Main functionality

Delegator contracts enable users to create precise delegations to other accounts, allowing these accounts to act on behalf of the delegator. These delegations can be finely tuned to accommodate a wide array of use cases.

Contracts wishing to utilize this functionality must inherit from `DeleGatorCore.sol` , thereby gaining access to the `executeDelegatedAction` function, which is exclusively callable by the `DelegationManager` .

The `DelegationManager` serves as a secondary entry point for the inheriting contract, endowed with full authority to execute actions on its behalf. It is responsible for validating delegation chains within the `redeemDelegation` function. These chains serve as proof that the actions to be executed by the inheriting contract align with the original delegator's intent, as the `redeemDelegation` function in `DelegationManager` can be invoked by anyone.

Delegation Chains

A delegation chain is constructed from `Delegation` structs, wherein the delegator's intent is defined and signed. The root `Delegation` includes a `ROOT_AUTHORITY` identifier, which designates the creator of the delegation as the account authorizing another account to act on its behalf. In subsequent delegations, the `authority` field is populated with the hash of the previous `Delegation` , enabling the creation of delegation chains.

In these chains, a `delegate` of an already established `Delegation` can create a new `Delegation` , wherein they assume the role of the `delegator` and assign an account as the new `delegate` . The `Delegator` can customize the new `Delegation` values to suit their desired requirements. The `authority` field plays a crucial role in the construction of `delegation` chains since it records the `Delegation` hash of the previous `Delegation` where the `delegator` of the new `delegation` was the `delegate` . The new `delegator` must sign this new `Delegation` to attest that the new delegate, Caveats, salt, and authority represent their intent, or validate the hash of the newly created `Delegation` on-chain.

To execute a call on behalf of the root `delegator` , a `delegate` that is mentioned in a valid `delegation` chain will call `redeemDelegation` providing the `action` he wants to execute on behalf of the root `delegator` , and a `Delegation` chain that links the caller as the `delegate` of the last `Delegation` to the root `Delegation` .

Enforcers and Caveats

To verify that the `action` provided by the caller adjusts to the terms stipulated by each `delegator` in the `delegation` chain, `DelegationManager` calls a series of hooks before and after the `action`. These hooks can call one or more `enforcer` contracts to validate if the `action` is valid according to the `Caveats` stipulated on each specific `Delegation`.

Each `Delegation` in the delegation chain can contain an array of `Caveats`, which include the enforcer address and the caveat `terms` and `arguments`. This gives the user the possibility of fine-tuning the desired outcome of the executed `action` on his behalf.

Delegation authorization

If a user prefers not to provide a signature for the `Delegation` hash, they can invoke the `delegate` function in the `DelegationManager` contract, supplying a `Delegation` where they are the `delegator`. This function stores the `Delegation` hash on-chain, eliminating the need for any entity to provide a valid signature to attest that it was authorized by the `delegator`.

To revoke the authority previously granted via a signed `Delegation` or an on-chain stored `Delegation`, a user can invoke the `disableDelegation` function, providing a `Delegation` where they are the `delegator`. This action stores the `Delegation` hash on-chain and blocks any further use of that `delegation`. The process can be reversed by calling `enableDelegation`.

Ultimately, the `DelegationManager` contract will call the `executeDelegatedAction` function from the root `delegator` and execute the `action` inputted by the caller.

3.2 DeleGatorCore.sol

The `DeleGatorCore` abstract contract provides inheriting contracts the capability to execute `Delegations` through the `DelegationManager` system. It also establishes a foundational structure to support ERC4337 functionality, including functions for depositing and withdrawing native tokens fund their account in the `entryPoint` contract. Additionally, it includes a series of modifiers that restrict access control for key functions to the `entryPoint` contract and itself, allowing for potential future implementations.

The objetive is for other contracts, such as `MultiSigDeleGator.sol` and `HybridDeleGator.sol`, to inherit from this contract and override the `isValidSignature` function to enable custom validation during `validateUserOp` for ERC4337 transactions.

3.3 MultiSigDeleGator.sol

`MultiSigDeleGator` extends the functionality of `DeleGatorCore`, overriding its `isValidSignature` to provide multi-signature capabilities to the contract. The contract also includes additional functions to manage the state of signers eligible to sign ERC4337 transactions and `Delegations`, and implements UUPS (Upgradeable Universal Proxy Standard) functionality.

3.4 HybridDeleGator.sol

`HybridDeleGator.sol` enhances the capabilities of `DeleGatorCore` by overriding its `isValidSignature` function to support multiple signature schemes. This allows users to create accounts that do not solely rely on the traditional ECDSA secp256k1 curve for public key recovery. Users can store one or more public keys, including P256 public keys, and sign ERC4337 transactions and `Delegations` using raw P256 signatures or WebAuthnP256 signatures.

The contract also includes additional functions to manage the state of the stored public keys and implements UUPS (Upgradeable Universal Proxy Standard) functionality.

4 Security Specification

4.1 MultiSigDelegator

Actors:

1. Entry Point:

The entry point contract represents a valid call from the owners of the MultiSig account(`signers`). If enough `signers` (compared to the `threshold`) have signed the ERC4337 transaction, the entry point will call the MultiSig account with the calldata provided in the ERC4337 transaction on behalf of the `signers`.

For this reason, functions with the `onlyEntryPointOrSelf` and `onlyEntryPoint` modifiers can not only manipulate the state of the MultiSig contract but also call other contracts on behalf of the MultiSig at will.

If the number of `signers` equal to the recorded `threshold` have signed the ERC4337 transaction, the entry point contract can:

- Execute any `Action` on behalf of the MultiSig (including calling `delegationManager.redeemDelegation` for other `Delegations` where the MultiSig is appointed as a `delegate`)
- Execute batches of transactions on behalf of the MultiSig
- Validate a `Delegation` on-chain (where the MultiSig is the `delegator`) so that the `delegate` doesn't have to provide the signatures that attest that the `Delegation` was authorized by the MultiSig signers
- Disable a `Delegation` on-chain, preventing other users with the `Delegation` and a valid array of signatures that attest that the `Delegation` was authorized by the MultiSig from using that signed `Delegation` in `delegationManager.redeemDelegation`.
- Change the implementation contract in the proxy, having the possibility of retaining the DeleGator storage or clearing it
- Add signers
- Remove signers
- Replace signers
- Update threshold
- Fund the MultiSig account in the entryPoint contract through `addDeposit` function
- Withdraw funding from the entry point contract through `withdrawDeposit` function
- Fund the MultiSig account with native tokens

2. `DelegationManager`:

The `DelegationManager` is an actor that represents an external user who has provided

- An `Action` to be executed by the MultiSig
- A `Delegation` chain that attests that the `Action` provided by the user is valid based on the `Caveats` that are imposed in the `Delegation` chain, and links the root delegator(MultiSig) and the leaf delegate(external user).

Under these circumstances, the DelegationManager will call the function `executeDelegatedAction` from the root `delegator` (multiSig), executing the `Action` provided on behalf of the MultiSig account.

Note: The `DelegationManager` is pausable, so this functionality is restricted to when the `DelegationManager` is `unpaused`.

3. External user:

- Fund the MultiSig account in the entryPoint contract through `addDeposit` function
- Fund the MultiSig account with native tokens.

> Note: Users can also send ERC721 and ERC1155 tokens through safeTransfer to the MultiSig since it implements `TokenCallbackHandler`.

4.2 HybridDeleGator:

Actors:

1. Entry Point:

The entry point contract represents a valid call from the owner of the account, based on the public keys registered in the `HybridDeleGator` contract. An ERC4337 transaction is considered valid if it recovers one of the public keys or the EOA registered in the `HybridDeleGator` contract, using the appropriate signature schemes.

For this reason, functions with `onlyEntryPointOrSelf` and `onlyEntryPoint` modifiers can not only manipulate the state of the `HybridDeleGator` contract but also call other contracts on behalf of the `HybridDeleGator` at will.

If the ERC4337 transaction is proven valid, the entry point contract can:

- Execute any action on behalf of the `HybridDeleGator` (including calling `delegationManager.redeemDelegation` for other `Delegations` where the `HybridDeleGator` is appointed as a `delegate`)
- Execute batches of transactions on behalf of the `HybridDeleGator`
- Validate a `Delegation` on-chain (where the `HybridDeleGator` is the delegator) so that the `delegate` doesn't have to provide a signature that attests that the `Delegation` was authorized by the `HybridDeleGator` owner
- Disable a `Delegation` on-chain, preventing other users with the `Delegation` and a valid signature that attests that the `Delegation` was authorized by the `HybridDeleGator` from using that signed `Delegation` in `delegationManager.redeemDelegation`
- Change the implementation contract in the proxy, having the possibility of retaining the DeleGator storage or clearing it
- Add keys
- Remove keys
- Update the keys and the owner
- Transfer ownership
- Renounce ownership
- Fund the `HybridDeleGator` account in the entryPoint contract through `addDeposit` function
- Withdraw funding from the entry point contract through `withdrawDeposit` function
- Fund the `HybridDeleGator` account with native tokens

2. `owner`:

The `owner` is an address recorded in storage, ideally an EOA to take advantage of its functionalities, that can sign transactions with Secp256k1 curve signature scheme. In `_isValidSignature`, if the signature bytes have a length of 65, it will try `ECDSA.recover` to verify that the 4337 transaction is indeed authorized by the `owner`. The `HybridDeleGator` expects the rest of the signatures to follow the Secp256r1 curve signature scheme.

3. `DelegationManager`:

The `DelegationManager` is an actor that represents an external user who has provided

- An `Action` to be executed by the `HybridDeleGator`
- A `Delegation` chain that attests that the `Action` provided by the user is valid based on the `Caveats` that are imposed in the `Delegation` chain, and links the root delegator(`HybridDeleGator`) and the leaf delegate(external user).

Under these circumstances, the DelegationManager will call the function `executeDelegatedAction` from the root `delegator` (`HybridDeleGator`), executing the `Action` provided on behalf of the `HybridDeleGator` account.

Note: The `DelegationManager` is pausable, so this functionality is restricted to when the `DelegationManager` is `unpaused`.

4. External user:

- Fund the `HybridDeleGator` account in the entry point contract through `addDeposit` function
- Fund the `HybridDeleGator` account with native tokens

> Note: Users can also send ERC721 and ERC1155 tokens through `safeTransfer` to the `HybridDeleGator` since it implements `TokenCallbackHandler`.

4.3 DelegationManager:

Security Properties

1. **Delegation Validation**:
   - Delegations can be cached on-chain to avoid having to provide a signature that validates a `Delegation` off-chain.
   - Delegations can be validated and executed through the `redeemDelegation` function.
   - Only the delegator can cache (`delegate`), disable (`disableDelegation`), or enable (`enableDelegation`) a delegation.
2. **Signature Verification**:
   - Off-chain delegations must be signed by the delegator and are validated at the time of execution.
   - The contract supports EOA (Externally Owned Account) and contract-based delegators.
   - For EOA delegators, the contract uses ECDSA signature recovery to verify the signature.
   - For contract-based delegators, the contract uses the `isValidSignature` function defined by the IERC1271 interface.
3. **Delegation Chain Validation**:
   - Delegations are ordered from leaf to root, with the last delegation in the array having the root authority.
   - Each delegation in the chain must be validated against its parent delegation, ensuring the authority and delegate are correctly set.
4. **Caveat Enforcement**:
   - Caveats are enforced in a specific order: `beforeHook` from leaf to root, execute action, `afterHook` from root to leaf.
   - Each caveat enforcer can add additional checks or restrictions before and after the action is executed.
5. **Modifiers**:
   - `onlyDeleGator`: Ensures that only the specified delegator can call certain functions.
   - `onlyOwner`: Ensures that only the contract owner can call certain administrative functions.
   - `whenNotPaused`: Ensures that functions can only be executed when the contract is not paused.
6. **Pause Functionality**:
   - The owner can pause and unpause the delegation redemption functionality using the `pause` and `unpause` functions.
   - When paused, the `redeemDelegation` function cannot be executed.

Actors:

1. `Owner`:

The owner of the contract is set during construction and can:

- Pause and Unpause the `redeemDelegation` function.
- Transfer Ownership of the `DelegationManager` contract to another address.
- Renounce Ownership, transferring the ownership to `address(0)`.

2. `delegator`:

Some functions have an `onlyDelegator` modifier that ensures these functions can only be called by the `delegator` specified in the `Delegation` provided as an argument. Under these circumstances, the `delegator` can:

- `delegate`, marking the `delegationHash_` on-chain as a valid delegation.
- `disableDelegation`, marking the `delegationHash_` on-chain as an invalid delegation.
- `enableDelegation`, reversing the process described in `disableDelegation`.

3. External user:

Any account can call `redeemDelegation` and execute an `Action` on behalf of the root `delegator`, if a valid `delegation` chain and a valid `action` are provided.

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 Important Points to Consider and Keep in Mind Medium Acknowledged

Description

Granting someone else control over one's own account – even if the intention is to have limitations in place – is an inherently dangerous operation. Hence, it is of crucial importance that users have a thorough understanding of the system, so they know exactly what they're doing. In the following, we list just a few points that might not be entirely obvious but are important to consider and keep in mind. We are not trying to achieve completeness but merely want to illustrate the importance of understanding the system in detail.

- Caveats limit what a delegator can do. This is achieved via enforcer contracts. Many of these enforcer contracts are low-level and of a technical nature. It is essential to understand what they do and don't achieve. Several findings below discuss subtle aspects of particular enforcers in more detail.

The behavior of an enforcer is controlled by the `terms` it is given – a low-level `bytes` sequence that is decoded into parameters. Obviously, getting these `terms` right is key to actually enforce the desired restrictions, but the low-level and enforcer-specific encoding makes that difficult to assess. Special care has to taken in crafting enforcer `terms`.
Finally, it is crucial to be aware of the fact that delegations are unrestricted by default: Unless caveats are added, everything is allowed!

- Delegations can be revoked at any time by the delegator – possibly even in a front-running manner. Hence, there is no guarantee for the redeemer that the action will be executed; the transaction can revert and the gas be lost. Revoking a delegation could either happen "regularly" via `disableDelegation` or, more subtly, by making the `isValidSignature` call fail (in case of a contract delegator) or by making a call to an enforcer fail (if the delegator has control over that). It should also be noted that this applies not only to the root delegator but to any delegator along a delegation chain.
- Offchain delegations are validated every time the delegation is redeemed, while onchain delegations remain valid until explicitly disabled. To illustrate this difference, let us consider a `MultiSigDeleGator` with Alice, Bob, and Carol as signers and `threshold` 2. Assume Alice and Bob both sign a delegation. This offchain delegation can be redeemed successfully as long as the configuration of the `MultiSigDeleGator` doesn't change. If, however, the `threshold` is increased to 3 or Bob and Carol kick Alice out of the multisig, future redemptions of this delegation will fail (until, perhaps, the configuration changes again). If, on the other hand, Alice and Bob create an onchain delegation, it stays valid even if the `threshold` is increased or Alice leaves the multisig; it can only be explicitly disabled.
- In a `MultiSigDeleGator`, the `threshold` is enforced for *delegating*, but a single party is sufficient for the redemption of a delegation. Consider, for instance, a `MultiSigDeleGator` with Alice and Bob as signers and `threshold` 2. Both Alice and Bob have to agree in order for a delegation to Carol to become valid, but after that, Carol alone can use the delegation to execute actions on the Alice-Bob multisig. Since that effectively bypasses the `threshold`, a reasonable delegation from a `MultiSigDeleGator` will, in most cases, include caveats.
- While they look very similar at the surface level, there is a crucial difference between root and non-root delegations. The root delegator delegates access to their *own* account, while further delegations along the chain just re-delegate the rights that have been granted, possibly adding further restrictions through caveats. From a different angle, this means that delegators have to make sure they're not accidentally signing a *root* delegation – and, therefore, access to their own account – when all they want to do is re-delegate another delegation. Root delegations have `authority` `0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF`, while for non-root delegations, the `authority` is the hash of the previous delegation.
- The redeemer of a delegation chain can (to the extent permitted by the caveats) freely choose the time of redemption and can also influence the chain state for a redemption or wait for a particular state that is considered favorable. Hence, by their very nature, delegations are susceptible to front-running, back-running, and sandwiching attacks by the redeemer.
- Delegators along a delegation chain can freely choose the contracts they use as "enforcers". Hence, just like the redeemer, they can also front- and back-run and launch sandwiching attacks. As a general rule, any delegator along a delegation chain can make arbitrary calls (with the right enforcers in place), including other delegation redemptions.

Recommendation

As mentioned above, this list is not meant to be exhaustive. It is crucial to understand the system in depth in order to be aware of these and other consequences of its design.

5.2 Delegators Can Abuse Gas for Their Own Purposes Medium Acknowledged

Resolution

Client's comment:

> We're aware of that and will keep an eye on it, but we're not making any changes for the time being.

Description

When a delegation is redeemed, delegators can abuse gas for their own purposes. This could happen in the following ways:

1. For contract delegators, their `isValidSignature` function is called during redemption. Since this is a `view` function, the compiler inserts a `STATICCALL`, which makes state changes impossible and, therefore, an attack less interesting. Gas *griefing*, i.e., just guzzling up all the gas to make the transaction revert, is well possible, though.
2. Delegators can freely choose the enforcer addresses for their caveats. It is conceivable that they add a caveat to their delegation with an "enforcer" that does useful, gas-intensive work for them. Gas griefing is possible too, but "stealing gas" for their own purposes is even more interesting, obviously.
3. Root delegators have more power than other delegators along a delegation chain. Even if they start with a regular `MultiSigDeleGator` or `HybridDeleGator` proxy, they could still upgrade to an arbitrary contract. Or they could directly start with an implementation of their own. Hence, they are in complete control of the root account and can do anything they want with the incoming call and gas. See also issue 5.3 for a related discussion.

Recommendation

(1) could be mitigated by sending only a fixed amount of gas with the `isValidSignature` call, but that would also limit honest signature verification and is, strictly speaking, not compliant with ERC-1271. One might try to tackle (2) with an enforcer allowlist, but that would limit the system's flexibility, and depending on the enforcer, there can still be variants of this attack via the `terms` and/or `args`. For instance, the `ERC20AllowanceEnforcer` makes a call to the token address given in the `terms`. In this case, it is a `STATICCALL`, but it is conceivable that a different enforcers might want to make state-changing calls to an address supplied in the `terms`.

It is the system's design and nature that many different parties are involved in the redemption of a delegation chain and are handed control over what's happening. Therefore, it is probably difficult and/or would require inconvenient restrictions that limit the system's versatility to get rid of this problem entirely. The countermeasures outlined in the previous paragraph could be considered, but they have limitations and/or downsides. At the very least, gas sent with a transaction should be closely monitored, and users should be aware of the abuse scenarios outlined above.

5.3 No Guarantee That Action Is Executed Even if Transaction Succeeds Medium Acknowledged

Resolution

Client's comment:

> We are aware of this and users can leverage afterHook of a caveat enforcer to validate actions.

Description

It is possible that a `redeemDelegation` call succeeds, all caveats along the delegation chain are processed successfully, but the action is not executed. That is because the root delegator could upgrade their account to a different implementation that silently ignores the `executeDelegatedAction` call from the delegation manager or does something else entirely. Or, without upgrading, an account contract that behaves in this way could be used right from the start.

This is, in and of itself, an important fact to be aware of, but it has also consequences for stateful enforcers because the state changes in them will persist in such a case.

Examples

- A contract deployed through the `DeployedEnforcer` will continue to exist – which is probably not a bad thing, except that the redeemer paid the gas for the transaction including this deployment without getting the action executed.
- The `ERC20AllowanceEnforcer` will increase the spent amount (i.e., reduce the remaining allowance) in such a case, even though the action with the transfer doesn't happen.
- Similarly, the `LimitedCallsEnforcer` will count the use of this delegation, and the `NonceEnforcer` will mark the nonce as used.

Recommendation

While this is definitely – and at the very least – something to keep in mind, the consequences with the enforcers that are in scope for this review don't seem too grave. However, for other enforcers, it may be a more serious problem, depending on their design.

The issue could be prevented as follows: The delegation manager approves certain implementation contracts (that execute the action as intended). Before calling `executeDelegatedAction` on the root delegator's account, the delegation manager verifies the following:

1. The codehash at the account address matches the only allowed proxy contract code.
2. The proxy uses an implementation contract that has been approved on the delegation manager. For this to work, a proxy contract has to be used (by all DeleGator accounts) that retrieves the implementation's address *with code on the proxy*. That is currently not the case. This solution would also make the system a bit more permissioned, as implementations would have to be approved by the owner of the delegation manager.

Remark

See also issue 5.2 for a related discussion.

5.4 `ERC20BalanceGteEnforcer` : Lock Mechanism Insufficient Medium

Description

The current implementation of the `ERC20BalanceGteEnforcer` contains a potential flaw when handling several `Delegations` from the same `delegator`. Specifically, if the `delegator` creates several `Delegations` in the delegation chain that call the `ERC20BalanceGteEnforcer` with the same `token`, the lock mechanism fails to prevent double counting of the token balance difference. This results in incorrect enforcement of the balance requirements, as the same balance difference might be counted for multiple delegations.

The issue arises because the locking mechanism only uses the delegation hash as key, while the balance caching uses the delegator and the token – and the latter can be the same for different delegations and, therefore, delegation hashes.

DeleGator/src/enforcers/ERC20BalanceGteEnforcer.sol:L15-L17

```
mapping(address delegationManager => mapping(address delegator => mapping(address token => uint256 balance))) public
    balanceCache;
mapping(address delegationManager => mapping(bytes32 delegationHash => bool lock)) public isLocked;
```

DeleGator/src/enforcers/ERC20BalanceGteEnforcer.sol:L26-L42

```
function beforeHook(
    bytes calldata _terms,
    bytes calldata,
    Action calldata,
    bytes32 _delegationHash,
    address _delegator,
    address
)
    public
    override
{
    require(!isLocked[msg.sender][_delegationHash], "ERC20BalanceGteEnforcer:enforcer-is-locked");
    isLocked[msg.sender][_delegationHash] = true;
    (, address token_) = getTermsInfo(_terms);
    uint256 balance_ = IERC20(token_).balanceOf(_delegator);
    balanceCache[msg.sender][_delegator][token_] = balance_;
}
```

DeleGator/src/enforcers/ERC20BalanceGteEnforcer.sol:L49-L64

```
function afterHook(
    bytes calldata _terms,
    bytes calldata,
    Action calldata,
    bytes32 _delegationHash,
    address _delegator,
    address
)
    public
    override
{
    delete isLocked[msg.sender][_delegationHash];
    (uint256 amount_, address token_) = getTermsInfo(_terms);
    uint256 balance_ = IERC20(token_).balanceOf(_delegator);
    require(balance_ >= balanceCache[msg.sender][_delegator][token_] + amount_, "ERC20BalanceGteEnforcer:balance-not-gt");
}
```

To illustrate with a concrete example:

1. A delegator creates a delegation (D1) with a caveat that executes calls to the `ERC20BalanceGteEnforcer` for a certain token. He is appointed as a delegate in another delegation in the delegation chain and creates another delegation (D2) with the same token as D1 with intentions to increase his balance by another amount stipulated in the terms.
2. The delegate of this second delegation (D2) executes `redeemDelegation` with the previously mentioned delegation chain.
3. The `beforeHook` of D2 (leaf to root) sets a lock using the delegation manager and D2's delegation hash, the token state is recorded in `balanceCache`.
4. The lock does not prevent D1 from executing because it considers the D2 delegation hash, creating a second lock based on D1's delegation hash and records `balanceCache` a second time. We assume no transfer has occurred yet, so the recorded balance does not actually change.
5. The action is executed and entails a token transfer to the delegator that is at least as high as the higher of the amounts requested in the two `ERC20BalanceGteEnforcer` terms. (For simplicity, we may assume they're both the same.)
6. The `afterHook` is executed from D1 (root to leaf) and the D1 lock is deleted. The balance after execution is compared with the `balanceCache` variable + the amount stipulated in the terms; the check passes.
7. The `afterHook` from D2 is executed, the D2 lock is deleted and the balance after execution is compared with the `balanceCache` variable + the amount stipulated in the terms again; this check passes too.

The balance change would therefore be counted twice, once for each delegation, leading to enforcement errors.

The example may seem a bit contrived because the same delegator appears twice in the delegation chain; however, there is nothing that prevents such delegation chains. Moreover, there are similar scenarios, based on reentrancy, that don't require this. For example, the action could redeem another delegation chain, which includes a delegation with the same delegator and token but different delegation hash.

So we have established that using the delegation hash as key for the lock is not sufficient, and we should instead use the same key as `balanceCache`, which is delegator and token – in addition to `msg.sender`, which is the delegation manager. Are we good then? Unfortunately, not! A similar "double count" scenario as above could arise for the same delegator and token but with different delegation managers. Unlike in the examples above, we would then set and check different entries in the `balanceCache` mapping (and have different locks, of course), but we would still accept the balance change for the delegator address twice. And just leaving the `msg.sender` out of the keys to solve this is not a good idea, as that would allow anyone to set a lock and never release it again.

Recommendation

We will address this problem together with another one in issue 5.5. The (somewhat limiting) solution will be to always use a fresh address to receive payments to in the `ERC20BalanceGteEnforcer`.

5.5 `ERC20BalanceGteEnforcer`: Separate Address Needed for Payment Medium

Description

The `ERC20BalanceGteEnforcer`, from a technical perspective, enforces that the delegator's balance (of a particular token given in the `terms`) has increased by a certain amount (that is also encoded in the `terms`) when the `afterHook` is executed compared to when the `beforeHoook` was executed. The probable intention is to make sure that some form of payment happened between `beforeHook` and and `afterHook`. That might be the intention to get paid for using the delegation, but the enforcer leaves that open, so let's just call it the "intended payment." Any other token transfer that occurs would be called a "side payment."

An example scenario for a side payment could be the following: The delegator has withdrawable funds in a contract that implements a pull pattern, where the address to transfer the funds to is fixed – in our case that would be the delegator's address – but anyone can execute the transfer. An actor with the intention to "fake the payment" would withdraw these funds to the delegator's address beween `beforeHook` and `afterHook`. Or there could be a contract the delegator has interacted with in the past that pays some form of reward, and the owner/admin of this contract (or whoever can initiate the reward payout) gets control of execution between `beforeHook` and `afterHook` and pushes out the reward; that could happen in an enforcer or in the action – directly or indirectly. These examples do not assume that the delegator is the root of the delegation chain, but if that is the case, it seems even more plausible that the action – which is executed from the root delegator's account – can, in some way or other, lead to a token transfer that the delegator would not see as intended payment.

Recommendation

While these scenarios might seem vague or far-fetched, there may very well be highly plausible ones we can't think of now. In any case, we believe that the argument shows that the enforcer – while it technically does what it promises, i.e., ensure that the delegator sees a certain balance increase between `beforeHook` and `afterHook` – can't ensure that the intention behind the use of this enforcer is met under all circumstances. Currently, it falls upon the delegator to use this enforcer only if they can guarantee that any transfer that could possibly happen between `beforeHook` and `afterHook` would be the intended payment. That can be difficult or impossible for a regularly used account.

Taking also into account the so far unresolved issue 5.4, we suggest the following: The `terms` should be extended with another address parameter `paymentAddress`. As the name suggests, this is the address the delegator expects the payment to. It should be a fresh address that has never been used before and is only used for this purpose, in this particular delegation, on this particular delegation manager. The user should be educated accordingly and instructed to provide a fresh address. Use cases that don't allow a freely chosen address, i.e., the delegator (as a person) expects a payment necessarily to their delegator address (as in the `Delegation` struct), should not be supported at all due to the risks described above and in issue 5.4.

One could go as far as requiring in the contract that the provided payment address is different from the delegator address to protect users from accidental misuse. However, just because the target address is different from the delegator address doesn't guarantee it's a fresh address in the sense above. So it's unclear how useful that would be.

Obviously, the balance checks shouldn't occur for the `delegator` anymore but for the new `paymentAddress`. The mappings should be defined as follows and used accordingly:

```
mapping(address delegationManager => mapping(address delegator => mapping(address token => mapping(address paymentAddress => u
mapping(address delegationManager => mapping(address delegator => mapping(address token => mapping(address paymentAddress => b
```

5.6 `AllowedCalldataEnforcer`: Must Only Be Used for Static Types Medium

Description

Several enforcers have a *technical* name and specification, but they're most likely used with a specific *intention* in mind. To make sure they don't just work on a technical level but fail to enforce the intention, certain usage restrictions might apply, certain preconditions might have to be met or some subtleties considered when they're used.

The `AllowedCalldataEnforcer`, from a technical perspective, enforces that a certain substring of the action's `data` part is a specific byte sequence. The start position in the `data`, the length of the substring, and the expected byte sequence are supplied as `terms` to the `beforeHook`.

DeleGator/src/enforcers/AllowedCalldataEnforcer.sol:L37-L47

```
uint256 dataStart_;
uint256 byteLength_;
bytes memory value_;
bytes memory calldataValue_;

(dataStart_, byteLength_, value_) = getTermsInfo(_terms);
require(dataStart_ + byteLength_ <= _action.data.length, "AllowedCalldataEnforcer:invalid-calldata-length");

calldataValue_ = _action.data[dataStart_:dataStart_ + byteLength_];

require(_compare(calldataValue_, value_), "AllowedCalldataEnforcer:invalid-calldata");
```

From a higher-level perspective, the typical use case is probably to restrict *function arguments* to specific values. For instance, assume we have a function whose first parameter is a `uint256`. Then we could utilize an `AllowedCalldataEnforcer` to restrict this argument to the number 42: The start position in the action's data would be 4 (bytes 0–3 are the function selector), the length would be 32 bytes, and we'd provide
0x000000000000000000000000000000000000000000000000000000000000002A as expected byte sequence.

Most likely, the `AllowedCalldataEnforcer` will be combined with an `AllowedTargetsEnforcer` to only allow calls to a specific address and an `AllowedMethodsEnforcer` to make sure we can only call a specific function on the target contract. To give a concrete example, with an `AllowedTargetsEnforcer`, an `AllowedMethodsEnforcer`, and an `AllowedCalldataEnforcer`, we can enforce that the action is a WETH transfer to a specific address. In this example, any transfer amount is allowed, but with an additional `AllowedCalldataEnforcer` we could also restrict the amount to 1 WETH, say. (The astute reader may have noticed that in this particular example, there are optimization possibilities because the two arguments are consecutive and directly follow the selector, so that we could get away with an `AllowedTargetsEnforcer` and a single `AllowedCalldataEnforcer`.)

However, there is a subtle point to how Solidity encodes and decodes data. While there is a standard encoding, called strict encoding, the Solidity ABI-decoder, which is responsible in a contract to decode the calldata into function arguments, accepts not only this strict encoding but is more lenient. We recommend studying the Contract ABI Specification for the details, but the gist is that the argument for a dynamic-type parameter *can't be found at a fixed place* in the calldata. And conversely, reading from a fixed place in the calldata does not necessarily give us what later becomes the argument in the function.

After we have given an example above what works, we will now see an example that doesn't. Assume we have a contract deployed at address A that implements the following functions :

```
function foo(uint256 num, string calldata str) external { ... } ,
function bar(address addr, uint256[] memory arr) public { ... }
```

Here, it is **not** possible to use an `AllowedTargetsEnforcer`, an `AllowedMethodsEnforcer`, and an `AllowedCalldataEnforcer` to ensure that the action is a call to address A of the function `foo` with the string “hello” for `str` (or that `str` starts with “hel” or that `str` has a certain length). Similarly, it is not possible to enforce that we're calling `bar` with an array of length 5 or with 42 as first element of the array. This is because `string` anf `uint256[]` are dynamic types. It is important to understand that these restrictions could be **completely bypassed**! On the other hand, enforcing that `num` is 42 in a `foo` call or that `addr` is a specific address is possible because `uint256` and `address` are static types.

Recommendation

To use the `AllowedCalldataEnforcer` correctly, it is crucial to understand these limitations and, more generally, the technical details of how ABI-encoding and -decoding works. Again, we refer to the Solidity documentation for a deep dive. A highly important lesson is that the `AllowedCalldataEnforcer` must only be used for static types.

Remark

We won't go into details, but for the sake of completeness, it should be mentioned that the goals we described above (for the not working examples) could still be achieved with a *sequence* of `AllowedCalldataEnforcers` that also enforces strict encoding, i.e., we'd "enforce the pointers" too. That is, however, tedious and error-prone, and – again – requires intricate knowledge of the details of ABI-encoding and -decoding. The important message of this finding is that the "straightforward" way does not work and can be completely bypassed.

5.7 `DeleGatorCore` : Implementation Contracts Should Not Support Interfaces and Not Accept Tokens Minor

Description

`DeleGator` accounts are proxies, and accounts of a particular type (such as `MultiSigDeleGator` or `HybridDeleGator`) share a common implementation. Every account type should inherit from `DeleGatorCore` . State-changing functions shouldn't be executable directly on the implementation contract. Calling `view` functions directly on the implementation makes often no sense and the information returned could be useless or even misleading and lead to unwanted results.

An example of this is the `supportsInterface` function in `DeleGatorCore` . This function is used to inform the caller whether the called contract supports certain functionalities:

It gives the correct result on the proxy, but the implementation should not claim to support any of these interfaces (with the possible exception of IERC165 – but that alone is fairly useless) in order to avoid misleading callers.

Similarly, the token callback handlers `onERC721Received` , `onERC1155Received` , and `onERC1155BatchReceived` (currently inherited from the imported `TokenCallbackHandler) should not claim to be able to handle these token types on the implementation – because that's not the case, and tokens sent to the implementation contract will be stuck.

Additionally, `addDeposit` could be mistakenly called by a user who expects to fund his account in the entry point, accepting native tokens that would eventually get stuck.

Recommendation

There are also other options, but a pragmatic solution to the issues outlined above is to add an `onlyProxy` modifier to `DeleGatorCore.supportsInterface` and `DeleGatorCore.addDeposit` , making these functions revert if they are called directly in the implementation contract.

A similar approach would work for the token callback handlers, but these are defined in `TokenCallbackHandler` from the AA repository, and these functions are not virtual. We therefore recommend:

1. Replace these handlers with the OpenZeppelin versions, i.e., import and inherit from ERC721Holder and ERC721Holder, and remove the import of and inheritance from the AA versions.
2. Override each handler with an `onlyProxy` modifier.
3. In the body of the overriding handlers, call the same function on `super` .

5.8 `MultiSigDeleGator` and `HybridDeleGator` : `reinitialize` Function Should Be Removed Minor

Resolution

Client's comment:

> Needed for account upgrading.

Description and Recommendation

`MultiSigDeleGator` and `HybridDeleGator` both have a `reinitialize` function; `reinitialize` is to be called after an upgrade to a new implementation contract, but it should be noted that it will be called *on the new contract*. Since the current versions of `MultSigDeleGator` and `HybridDeleGator` are the first one, respectively, i.e., we're not upgrading from a previous version, there is currently no need for a `reinitialize` function, and they can both be removed. In fact, they *should* be removed because some users might call them accidentally, causing unnecessary complications for future updates.

If your intention was to offer users a way to prevent future upgrades of their proxy by setting the version number to `type(uint64).max` , a dedicated function solely for that purpose and with a descriptive name should be added instead.

5.9 `HybridDeleGator` : `_updateSigners` Reverts Overzealously Minor

Description

The internal `_updateSigners` function in `HybridDeleGator` takes an owner, a list of public keys, and a boolean argument and updates the owner as well as the signers list of the DeleGator account. The boolean argument `_deleteP256Keys` dictates whether the public keys should just be *added* to the existing set (if `_deleteP256Keys` is false) or *replace* the existing set (if `_deleteP256Keys` is true). As there is only one owner, it is always replaced. But the new owner can be the same as the old one, or the new owner can be `address(0)` , which means there is no owner and only signers.

A situation we want to avoid is ending up without anyone who can control the SCA, i.e., whenever we make a change to the owner and/or signers, we want to make sure that after the operation, there is at least one signer or the owner is not zero. That is achieved with the following line, where `keysLength_` is the number of public keys given to the function.

DeleGator/src/HybridDeleGator.sol:L300

```
if (_owner == address(0) && keysLength_ == 0) revert SignersCannotBeEmpty();
```

However, this check reverts overzealously because it doesn't take into account that `_deleteP256keys` might be `false`; in this case, we wouldn't *replace* signers – we would just *add* them. And adding no new signers, while setting the owner to zero is fine as long as the current list of signers is not empty.

Recommendation

We should only revert here only if – in addition to `_owner == address(0) && keysLength_ == 0)` being true – `_deleteP256Keys` is true or the current list of signers is empty.

5.10 `MultiSigDeleGator`: No Function to Retrieve the Number of Signers Directly Minor

Description

`MultisigDeleGator`, while it has a `view` function `getSigners` to retrieve all current signers, does not have a `view` function to just obtain *the number of signers*. Of course, it is possible to extract that information from the result of `getSigners`, but that can be inefficient since the entire array is read from storage. It should also be noted that `HybridDeleGator` has a `getKeyIdHashesCount` function in addition to `getKeyIdHashes`, so it seems likely that such a function has just been forgotten in `MultiSigDeleGator`.

Recommendation

Add a `getSignersCount` function to `MultiSigDeleGator` that returns the current number of signers.

5.11 `MultiSigDeleGator`: Caution Regarding `_addSignersAndThreshold` in the Next Version(s) Minor

Description

`DeleGatorCore` offers two variants of the `upgradeToAndCall`-type function: The first one, with the (normal) name `upgradeToAndCall`, first clears the DeleGator's storage (via the `_clearDeleGatorStorage` function, supposed to be implemented in concrete DeleGator Account implementations) and then calls `upgradeToAndCall` on OpenZeppelin's `UUPSUpgradeable` contract, from which `DeleGatorCore` inherits. The second function, with the name `upgradeToAndCallAndRetainStorage`, skips the `_clearDeleGatorStorage` call and does the same otherwise. The idea here is to offer a version with full flexibility to modifiy the proxy's storage during an upgrade and a more lightweight version for minor upgrades that don't need (significant) storage changes.

We discussed in issue 5.8 that the `reinitialize` functions should be removed in the first version of `MultiSigDeleGator` and `HybridDeleGator`, respectively. However, their current implementation indicates that the internal functions to be called for reinitialization will likely be `_addSignersAndThreshold` (for `MultiSigDeleGator`) and `_updateSigners` (for `HybridDeleGator`). While `_updateSigners` takes a boolean argument `_deleteP256Keys` that determines whether the existing signers should be removed before the new ones are added, `_addSignersAndThreshold` deletes the existing signers *unconditionally*. Hence, `_addSignersAndThreshold`, in its current form, wouldn't be compatible with `upgradeToAndCallAndRetainStorage`, which intentionally leaves the current signers untouched.

Recommendation

This is not a problem in and for the current version of the contract (and the `reinitialize` functions should be removed, as mentioned in issue 5.8). It should, however, be considered for the next version(s). Possibly, a boolean argument for `_addSignersAndThreshold` – as in `HybridDeleGator._updateSigners` – will be useful.

Regardless of this particular concern, the new versions of the contracts will have to be reviewed in their entirety, not only but also with respect to their suitability for upgrades from the current version.

5.12 `DelegationManager`: 2-Step Ownership Transfer and Renouncing Ownership Minor

Description

`Ownable` **vs.** `Ownable2Step`**:**

The `DelegationManager` has a contract owner whose privilege it is to pause and unpause delegation redemptions. Ownership is implemented via OpenZeppelin's `Ownable`, which is a well-established solution for the task at hand. The downside of `Ownable` is that is relatively easy to leave the contract unintentionally without owner access by accidentally transferring ownership to a wrong address.

Therefore, an improved version – `Ownable2Step` – has been devised that implements a 2-step ownership transfer. More specifically, ownership is first tentatively transferred to a new address by the current owner, and the new owner has to accept ownership first before the transfer is concluded. Until this happens, the old owner is still in control, has all owner privileges the contract offers, and can abort the transfer or choose a different address to tentatively transfer ownership to. This pattern reduces the risk of accidentally losing owner access to the contract.

Renouncing Ownership:

Renouncing ownership when the contract is paused means retiring the contract and ensuring that delegations can't ever be redeemed again. Unless this happens accidentally, that might be a desirable effect if, for instance, a critical bug in `redeemDelegation` is found. In that case, users can be sure that redemptions are not possible anymore and will never become possible again.

Renouncing ownership when the contract is not paused means it can never be paused (again). There is probably little or even no use for that, and it would prevent retiring the contract as described in the previous paragraph.

Recommendation

1. Consider replacing `Ownable` with `Ownable2Step`.
2. Renouncing contract ownership is always a dangerous operation and should only be done after very careful consideration. In particular, pay close attention whether the contract is paused or not.

5.13 `DeployedEnforcer` : Possibility for Better Protection Against Usage Mistakes Minor

Description

The intention of the `DeployedEnforcer` is to make sure that a certain contract has been deployed to a specific address. The target address and the calldata for a factory call comprise the terms for this enforcer; contrary to what the NatSpec annotation says, the factory address is not part of the terms but supplied as constructor argument.

The enforcer employs a codesize-based check to determine if the contract “is already there” or, if not, that it has been successfully deployed by the factory. Hence, the underlying assumption is: If there is code at all, it is the *right* code. For this to work reliably, the target address has to be a CREATE2 address (or we’d have to take a closer look at the factory contract) because with CREATE2, the deployment address is tied to the code to be deployed.

In summary, the enforcer works reliably if used correctly (i.e., with a CREATE2 address), but in and of itself, it does not guarantee that the target address has the *right* code. A simple example: Assume the user gets the target address in the `terms` wrong and accidentally chooses an address that happens to have code but is not the right contract. In that case, the enforcer hook would return successfully. While all enforcers rely on being used correctly, the `DeployedEnforcer` could offer better protection against usage mistakes by employing *codehash*-based checks instead of *codesize*-based ones. More specifically, the `terms` would also have to contain the codehash that is *expected* at the target address, and the *actual* codehash there should match the expectation.

Recommendation

There seems to be little disadvantage to adopting the codehash-based approach we outlined above, so we suggest considering that. Independently and generally, enforcer assumptions should always be properly documented. In this case, this includes that the target address has to be a CREATE2 address as well as the factory requirements.

5.14 `SimpleFactory` : Low-Level Code and Empty Revert Minor

Description

The `deploy` function in `SimpleFactory` uses assembly for deployment via `create2` :

DeleGator/src/utils/SimpleFactory.sol:L17-L20

```
assembly {
    addr_ := create2(0, add(_bytecode, 0x20), mload(_bytecode), _salt)
    if iszero(extcodesize(addr_)) { revert(0, 0) }
}
```

While there is no high-level Solidity construct for this task, OpenZeppelin’s `Create2` library provides a convenient wrapper that has been reviewed and tested and can be used to avoid having to deal with low-level code yourself.

Moreover, if the deployment fails, there is an assembly `revert(0, 0)` , which is essentially an “empty” revert. That is (1) not very descriptive and (2) not compatible with the reverts Solidity normally generates (i.e., `Error` and `Panic`). (While it *is* possible to generate an empty revert in Solidity with `revert()` , that is, in our experience, not often used in practice.)

Recommendation

In general, it makes sense to avoid using low-level code when possible. In this particular case, we recommend utilizing OpenZeppelin’s `Create2` library.

It should be noted that there is a small functional difference: OpenZeppelin’s `Create2.deploy` can succeed even if the deployment address does not end up having code (but the deployment was still successful), while the current version of `SimpleFactory.deploy` will revert in that case. Whether this difference is important depends on how exactly you want to use the factory. If you want to make sure indeed that not only the deployment was successful but that you really end up with code at the deployment address, you can still use `Create2.deploy` and check later if the returned address has code.

5.15 `HybridDeleGator` Should Only Be Deployed on Chains That Support P256 Signature Verification at `0xc2b78104907F722DABAc4C69f826a522B2754De4` Minor

Description and Recommendation

To ensure smooth operation, the `HybridDeleGator` contract should only be deployed on chains that support secp256r1 signature verification – as specified in EIP-7212 – at address `0xc2b78104907F722DABAc4C69f826a522B2754De4` , either via contract or via precompile at that address. If that is not the case, secp256r1 signature verification will always fail in the current codebase.

While this problem can be fixed by deploying the verifier contract later, users might get confused or unsettled by the permanently failing signature verification, so it’s best to avoid such a situation in the first place.

Ideally, this check would be part of the deployment script. It is, however, not sufficient to just check if the codesize at the address above is non-zero, since addresses with precompiles have codesize 0. Hence, a functional check is necessary, i.e., can a valid signature actually be verified?

5.16 Missing Events Minor

Description

It is well-known that events are not important *for* the contracts (and can, in fact, not even be read by a contract), but they’re important to keep “the outside world” informed about state changes and, well, events on the contracts. In general, state-changing functions should emit an event to have an audit trail and enable monitoring of smart contract usage.

The DeleGator codebase is a bit inhomogeneous with respect to events. While some contracts such as `MultiSigDeleGator` , `HybridDeleGator` (excluding the inherited `DeleGatorCore` in both cases), are well-equipped with events, the state-changing enforcers show a mixed picture with the occasional event missing or unindexec parameters that should be indexed. The `DeleGationManager` has, one the one hand, events for storing delegations onchain as well as for disabling and enabling delegations, but it does not emit an event when a delegation is redeemed – arguably the most central piece of functionality the contract offers. Finally, `DeleGatorCore`, the contract all DeleGator account implementations should inherit from, lacks events for important operations such as executing an action, deposits, withdrawals, etc.

Recommendation

We recommend a comprehensive and careful internal review of the entire codebase with respect to events. Determine which events are missing; these should then be implemented and tested. Make sure event parameters are `indexed` appropriately. For important parameter changes, it can make sense to emit both the old and the new value.

5.17 `DeleGatorCore` and Related Contracts: Miscellaneous Minor Points Minor

`DeleGatorCore`

A1. Unnecessary `return` in functions:

The following functions have a return statement but lack a return type declaration in their definition. Since there is nothing to return, the `return` statement is unnecessary:

DeleGator/src/DeleGatorCore.sol:L228-L230

```
function delegate(Delegation calldata _delegation) public onlyEntryPointOrSelf {
    return delegationManager.delegate(_delegation);
}
```

DeleGator/src/DeleGatorCore.sol:L239-L241

```
function disableDelegation(Delegation calldata _delegation) public onlyEntryPointOrSelf {
    return delegationManager.disableDelegation(_delegation);
}
```

DeleGator/src/DeleGatorCore.sol:L251-L253

```
function enableDelegation(Delegation calldata _delegation) public onlyEntryPointOrSelf {
    return delegationManager.enableDelegation(_delegation);
}
```

A2. Incorrect NatSpec annotations:

The NatSpec annotations for `disableDelegation` and `enableDelegation` say that “Delegations must be stored onchain to be enabled/disabled” and that these methods “must be executed through a UserOp”.

DeleGator/src/DeleGatorCore.sol:L232-L238

```
/**
 * @notice Disables a delegation from being used
 * @dev Delegations must be stored onchain to be enabled/disabled
 * @dev This method can only be called by the entry point contract or the contract itself, so it must be executed through a
 * UserOp
 * @param _delegation The delegation to be stored
 */
```

DeleGator/src/DeleGatorCore.sol:L243-L250

```
/**
 * @notice Enables a delegation to be used
 * @dev Delegations only need to be enabled if they have been disabled
 * @dev Delegations must be stored onchain to be enabled/disabled
 * @dev This method can only be called by the entry point contract or the contract itself, so it must be
 * executed through a UserOp
 * @param _delegation The delegation to be stored
 */
```

Both are not correct. Offchain delegations can be enabled and disabled too, and these two methods can also be executed (indirectly, via `executeDelegatedAction` and call to self) from the delegation manager, so not necessarily through a UserOp.

A3. The `onlySelf` modifier is never used:

DeleGator/src/DeleGatorCore.sol:L70-L73

```
modifier onlySelf() {
    if (msg.sender != address(this)) revert NotSelf();
    _;
}
```

`IDeleGatorCoreFull`

B1. Parameter names don’t start with underscore:

DeleGator/src/interfaces/IDeleGatorCoreFull.sol:L50

```
function withdrawDeposit(address payable withdrawAddress, uint256 withdrawAmount) external;
```

`ExecutionLib`

C1. `executeBatch` silently succeeds if given an empty array:

If that’s unwanted, it should revert in this case.

DeleGator/src/DeleGatorCore.sol:L157-L159

```
function executeBatch(Action[] calldata _actions) external onlyEntryPointOrSelf {
    ExecutionLib._executeBatch(_actions);
}
```

DeleGator/src/libraries/ExecutionLib.sol:L35-L39

```
function _executeBatch(Action[] calldata _actions) internal {
    for (uint256 i = 0; i < _actions.length; ++i) {
        _execute(_actions[i]);
    }
}
```

C2. Explicit conversion is not necessary:

DeleGator/src/utils/Types.sol:L49-L53

```
struct Action {
    address to;
    uint256 value;
    bytes data;
}
```

In `execute`, `_action.to` is already an `address`, therefore the explicit conversion to `address` is not necessary.

DeleGator/src/libraries/ExecutionLib.sol:L20

```
(bool success_, bytes memory errorMessage_) = address(_action.to).call{ value: _action.value }(_action.data);
```

C3. Calldata value `_actions.length` could be cached in a local variable.

DeleGator/src/libraries/ExecutionLib.sol:L35-L39

```
function _executeBatch(Action[] calldata _actions) internal {
    for (uint256 i = 0; i < _actions.length; ++i) {
        _execute(_actions[i]);
    }
}
```

5.18 `MultiSigDeleGator` : Miscellaneous Minor Points Minor

A. There should be a comment to clearly state the invariant:

`0 < threshold <= number of signers <= MAX_NUMBER_OF_SIGNERS`

B. Struct definition should be preceded by the NatSpec annotation:

`/// @custom:storage-location erc7201:DeleGator.MultiSigDeleGator`

DeleGator/src/MultiSigDeleGator.sol:L11-L15

```
struct MultiSigDeleGatorStorage {
    mapping(address => bool) isSigner;
    address[] signers;
    uint256 threshold;
}
```

Cf. ERC-7201.

C. Unnecessary check in `removeSigner` function:

DeleGator/src/MultiSigDeleGator.sol:L173

```
if (storedSignersLength_ == 1 || storedSignersLength_ == s_.threshold) revert InsufficientSigners();
```

The check `storedSignersLength_ == 1` is not necessary because we maintain the invariant:

`0 < threshold <= number of signers <= MAX_NUMBER_OF_SIGNERS`

This means if `number of signers == 1`, then we have `0 < threshold <= 1`, which implies `threshold == 1`. Hence, `number of signers == threshold`, so the second condition is `true` too.

D. Misleading function name:

The function name `_addSignersAndThreshold` invites mistakes because it suggests that the signers are simply *added*, but in reality the existing signers are removed first. Something like `_setSignersAndThreshold` seems less prone to misunderstandings. Moreover, the preceding NatSpec annotation is wrong; it says that the function "optionally deletes the current signers", but they are always deleted.

DeleGator/src/MultiSigDeleGator.sol:L240-L245

```
/**
 * @notice Adds the signers and threshold, optionally deletes the current signers
 * @param _signers List of new signers of the MultiSig
 * @param _threshold The new threshold of required signatures
 */
function _addSignersAndThreshold(address[] calldata _signers, uint256 _threshold) internal {
```

See also issue 5.11 for a related discussion.

E. Unnecessary read from storage:

It is not necessary to read the threshold from storage for the event.

DeleGator/src/MultiSigDeleGator.sol:L267-L268

```
s_.threshold = _threshold;
emit UpdatedThreshold(s_.threshold);
```

It is directly available as function argument `_threshold`.

F. Using `uint8` instead of `uint256` consumes more gas:

In `_isValidSignature`, type `uint8` for `validSignatureCount_` and `i` does not offer any advantage over `uint256`.

DeleGator/src/MultiSigDeleGator.sol:L304-L306

```
uint8 validSignatureCount_ = 0;

for (uint8 i = 0; i < signatureCount_; ++i) {
```

G. Storage variable is read repeatedly in a for loop:

DeleGator/src/MultiSigDeleGator.sol:L315

```
if (validSignatureCount_ >= s_.threshold) {
```

The threshold shouldn't be read repeatedly from storage, especially since this expression will always be false except in the last iteration of the loop, i.e., when `i == signatureCount_ - 1` (if it is reached).

H. `_initialize` function is not necessary:

DeleGator/src/MultiSigDeleGator.sol:L339-L341

```
function _initialize(address[] calldata _signers, uint256 _threshold) private {
    _addSignersAndThreshold(_signers, _threshold);
}
```

`_addSignersAndThreshold` could be called directly in `initialize`:

DeleGator/src/MultiSigDeleGator.sol:L95-L97

```
function initialize(address[] calldata _signers, uint256 _threshold) public initializer {
    _initialize(_signers, _threshold);
}
```

And `reinitialize` should be removed. Cf. issue 5.8.

I. Consider Using `EnumerableSet`:

OpenZeppelin's `EnumerableSet` is a data structure that is well suited to manage the set of signers: It supports addition, removal, number of elements (all in O(1)) and enumeration. The advantage compared to the current implementation is that removals don't require iteration and that less low-level code is needed.

5.19 `HybridDeleGator`: Miscellaneous Minor Points Minor

A. Unused imports can be removed:

DeleGator/src/HybridDeleGator.sol:L6

```
import { MessageHashUtils } from "@openzeppelin/contracts/utils/cryptography/MessageHashUtils.sol";
```

DeleGator/src/libraries/P256VerifierLib.sol:L6

```
import { Base64URL } from "../external/Daimo/utils/Base64URL.sol";
```

B. The struct definition should be preceded by the NatSpec annotation:

`/// @custom:storage-location erc7201:DeleGator.HybridDeleGator`

DeleGator/src/HybridDeleGator.sol:L15-L19

```
struct HybridDeleGatorStorage {
    address owner;
    mapping(bytes32 keyIdHash => P256PublicKey) authorizedKeys;
    bytes32[] keyIdHashes;
}
```

Cf. ERC-7201.

C. Errors:

DeleGator/src/HybridDeleGator.sol:L48-L52

```
/// @dev Error emitted when a P256 key already exists
error AlreadyExists(bytes32 keyIdHash, string keyId);

/// @dev Error emitted when a P256 key is not stored and attempted to be removed
error KeyDoesNotExist(bytes32 keyIdHash);
```

It is unclear why the `AlreadyExists` has the key ID hash as well as the full key ID as parameter, while the otherwise symmetric error `KeyDoesNotExist` just takes the hash. (Also the naming is slightly inconsistent, and `AlreadyExists` could be renamed to `KeyAlreadyExists` .)

D. Incorrect NatSpec annotations:

The NatSpec annotations for `initialize` and `reinitialize` suggest that it is not impossible to have a *contract* as `owner` for the `HybridDeleGator` :

DeleGator/src/HybridDeleGator.sol:L81-L82

```
* @dev The owner SHOULD be an EOA. Contract owners require staking in the EntryPoint to enable signature
* verification during UserOp validation.
```

However, `_isValidSignature` expects the `owner` to be an EOA, without exceptions:

DeleGator/src/HybridDeleGator.sol:L356

```
if (ECDSA.recover(_hash, _signature) == owner()) return ERC1271Lib.EIP1271_MAGIC_VALUE;
```

E. Redundant function call for value retrieval:

As `removeKey` (and comparable functions) normally read directly from storage instead of employing public functions to read storage values, it would probably be more consistent and gas-efficient to replace `owner()` with `s_.owner` in the following line:

DeleGator/src/HybridDeleGator.sol:L154

```
if (keyIdHashesCount_ == 1 && owner() == address(0)) revert CannotRemoveLastSigner();
```

F. Ownership can be transferred to `address(0)` :

OpenZeppelin's `Ownable.transferOwnership` functions reverts if the new owner is `address(0)` ; in order to renounce ownership of the contract, `Ownable` offers the `renounceOwnership` function instead. `HybridDeleGator` also offers a `transferOwnership` and a `renounceOwnership` function, but `transferOwnership` doesn't revert if the new owner is zero. In order to more closely mimic the well-known OZ functions, it might make sense for `transferOwnership` to revert too if `_newOwner == address(0)` . (It should be noted, though, that OZ's `transferOwnership` most likely reverts in order to prevent *accidental* renouncements. This is less of a danger in `HybridDeleGator` because ownership can only be renounced if there is at least one signer left that can still control the account.)

G. Inefficient string emptiness check:

`_addKey` : In order to determine whether the input string is empty, it might be simpler and more natural to check if its length is zero than to compare its hash with the hash of the empty string.

DeleGator/src/HybridDeleGator.sol:L257-L258

```
bytes32 keyIdHash_ = keccak256(abi.encodePacked(_keyId));
if (keyIdHash_ == keccak256(abi.encodePacked(""))) revert InvalidEmptyKey();
```

Note that `bytes(_keyId).length` gives us the length of `_keyId` .

H. Unnecessary use of assembly:

It is not necessary to use assembly in order to read the first word of the `bytes calldata` argument `_signature` .

DeleGator/src/HybridDeleGator.sol:L362-L365

```
bytes32 keyIdHash_;
assembly {
    keyIdHash_ := calldataload(_signature.offset)
}
```

Instead, an array slice could be used – as all the enforcers do to decode the terms.

I. Consider Using `EnumerableSet`:

OpenZeppelin's `EnumerableSet` is a data structure that is well suited to manage the set of key ID hashes: It supports addition, removal, number of elements (all in O(1)) and enumeration. The advantage compared to the current implementation is that removals don't require iteration and that less low-level code is needed.

5.20 `DelegationManager` and Related Contracts: Miscellaneous Minor Points Minor

`DelegationManager`

A1. The NatSpec markers for constructor and modifier have been transposed:

DeleGator/src/DelegationManager.sol:L69-L74

```
////////////////////////////// Modifier //////////////////////////////

/**
 * @notice Initializes Ownable and the DelegationManager's state
 * @param _owner The initial owner of the contract
 */
```

A2. OpenZeppelin's EIP712 library could be used for the domain separator computation:

DeleGator/src/DelegationManager.sol:L75-L79

```
constructor(address _owner) Ownable(_owner) {
    CHAIN_ID = block.chainid;
    DOMAIN_HASH = ERC712Lib.getEIP712DomainHash(NAME, DOMAIN_VERSION, CHAIN_ID, address(this));
    emit SetDomain(DOMAIN_HASH, NAME, DOMAIN_VERSION, CHAIN_ID, address(this));
}
```

DeleGator/src/DelegationManager.sol:L256-L259

```
function getDomainHash() public view returns (bytes32) {
    if (CHAIN_ID == block.chainid) return DOMAIN_HASH;
    return ERC712Lib.getEIP712DomainHash(NAME, DOMAIN_VERSION, block.chainid, address(this));
}
```

`IDelegationManagerMinimal`

B1. It's unclear why this minimal interface is needed:

It should be sufficient and simpler to just have a single interface `IDelegationManager`.

`EncoderLib`

C1. Cyclic Import:

DeleGator/src/libraries/EncoderLib.sol:L6-L12

```
import { EncoderLib } from "./EncoderLib.sol";

/**
 * @dev Provides implementations for common utility methods for Delegation.
 * @title Delegation Utility Library
 */
library EncoderLib {
```

5.21 Enforcers: Miscellaneous Minor Points. Minor

General:

A1. ERC-165 unused:

Enforcers implement ERC-165. While there is no harm in that except higher deployment costs, the motivation for that is unclear since no contract in the DeleGator codebase makes use of that. Making a `supportsInterface` call in the `DelegationManager` would only have a marginal advantage (and increased gas costs due to an extra call).

A2. Unused `args` and `redeemer_`:

Currently, no enforcer hook makes use of the caveat's `args` or the `redeemer_`. Hence, these parameters are not used right now and could be removed. We assume, however, that this is known and they are kept intentionally for (potential) future use.

A3. Consider using ABI-encoding:

Many enforcers expect a low-level encoding for the `terms`; usually, the data that makes up the terms consists of a few fixed-length elements and these are just concatenated without padding. Most enforcers then have a `getTermsInfo` function, which decodes the `bytes` terms into the actual argument.

Calldata array slices and the simple structure of the `terms` make this approach tolerable and quite readable, while also minimizing calldata length. Nevertheless, this is an ad hoc approach with a dedicated encoding for each enforcer – while Solidity's builtin ABI-encoding and ABI-decoding can be considered the standard method for encoding/decoding data.

Using this standardized way instead of self-written low-level code is worth consideration – although it should be noted that calldata size will increase with this approach.

`AllowedCalldataEnforcer`

B1. `byteLength_` doesn't have to be part of the `terms`:

It could be inferred from the length of `value_`. What's needed as `terms` is the byte sequence `value_` and at what offset in the action's data it is supposed to start.

DeleGator/src/enforcers/AllowedCalldataEnforcer.sol:L42-L47

```
(dataStart_, byteLength_, value_) = getTermsInfo(_terms);
require(dataStart_ + byteLength_ <= _action.data.length, "AllowedCalldataEnforcer:invalid-calldata-length");

calldataValue_ = _action.data[dataStart_:dataStart_ + byteLength_];

require(_compare(calldataValue_, value_), "AllowedCalldataEnforcer:invalid-calldata");
```

`AllowedMethodsEnforcer`

C1. Local variable `termsLength` doesn't end with an underscore.

DeleGator/src/enforcers/AllowedMethodsEnforcer.sol:L50-L51

```
uint256 termsLength = _terms.length;
require(termsLength % 4 == 0, "AllowedMethodsEnforcer:invalid-terms-length");
```

C2. Possible Panic exception:

In `beforeHook`, If the length of `_action.data` is less than 4, `_action.data[0:4]` will throw a Panic exception.

DeleGator/src/enforcers/AllowedMethodsEnforcer.sol:L32

```
bytes4 targetSig_ = bytes4(_action.data[0:4]);
```

This doesn't cause any inherent risk, but according to the Solidity documentation "Properly functioning code should never create a Panic". Hence, the function should first make sure that `_action.data.length` is at least 4 and revert otherwise.

C3. Note:

If the action's target contract (`to`) doesn't have a function with this selector but a fallback function, then the fallback function will be executed.

`AllowedTargetsEnforcer`

D1. `_terms.length` could be cached in a local variable `termsLength_`.

DeleGator/src/enforcers/AllowedTargetsEnforcer.sol:L48-L56

```
function getTermsInfo(bytes calldata _terms) public pure returns (address[] memory allowedTargets_) {
    uint256 j = 0;
    require(_terms.length % 20 == 0, "AllowedTargetsEnforcer:invalid-terms-length");
    allowedTargets_ = new address[](_terms.length / 20);
    for (uint256 i = 0; i < _terms.length; i += 20) {
        allowedTargets_[j] = address(bytes20(_terms[i:i + 20]));
        j++;
    }
}
```

`BlockNumberEnforcer`

E1. NatSpec is wrong/misleading:

DeleGator/src/enforcers/BlockNumberEnforcer.sol:L14-L19

```
/**
 * @notice Allows the delegator to specify the latest block the delegation will be valid.
 * @dev This function enforces the block number range before the transaction is performed.
 * @param _terms A bytes32 blocknumber range where the first half of the word is the earliest the delegation can be used and
 * the last half of the word is the latest the delegation can be used.
 */
```

1. L15: It's not only a before-limit, it's also an after-limit, i.e., a range.
2. L17-18: Suggest that the block numbers that constitute the range are *inclusive* for validity, but they're not:

DeleGator/src/enforcers/BlockNumberEnforcer.sol:L25

```
require(block.number > blockAfterThreshold_, "BlockNumberEnforcer:early-delegation");
```

DeleGator/src/enforcers/BlockNumberEnforcer.sol:L30

```
require(block.number < blockBeforeThreshold_, "BlockNumberEnforcer:expired-delegation");
```

The same is true for lines 37 and 38:

DeleGator/src/enforcers/BlockNumberEnforcer.sol:L37-L38

```
* @return blockAfterThreshold_ The earliest block number the delegation can be used.
* @return blockBeforeThreshold_ The latest block number the delegation can be used.
```

`TimestampEnforcer`

F1. NatSpec is wrong/misleading:

DeleGator/src/enforcers/TimestampEnforcer.sol:L15-L19

```
/**
 * @notice Allows the delegator to specify the latest timestamp the delegation will be valid.
 * @param _terms - A bytes32 timestamp range where the first half of the word is the earliest the delegation can be used and the
 * last half of the word is the latest the delegation can be used.
 */
```

1. L16: It's not only a before-limit, it's also an after-limit, i.e., a range.
2. L17-18: Suggest that the block numbers that constitute the range are *inclusive* for validity, but they're not:

DeleGator/src/enforcers/TimestampEnforcer.sol:L25

```
require(block.timestamp > timestampAfterThreshold_, "TimestampEnforcer:early-delegation");
```

DeleGator/src/enforcers/TimestampEnforcer.sol:L30

```
require(block.timestamp < timestampBeforeThreshold_, "TimestampEnforcer:expired-delegation");
```

The same is true for lines 37 and 38:

DeleGator/src/enforcers/TimestampEnforcer.sol:L37-L38

```
* @return timestampAfterThreshold_ The earliest timestamp the delegation can be used.
* @return timestampBeforeThreshold_ The latest timestamp the delegation can be used.
```

`DeployedEnforcer`

G1. Deviation from own style guide:

DeleGator/src/enforcers/DeployedEnforcer.sol:L4-L6

```
import { CaveatEnforcer } from "./CaveatEnforcer.sol";
import { Action } from "../utils/Types.sol";
import { Address } from "@openzeppelin/contracts/utils/Address.sol";
```

"The imports should be sorted by external dependencies an empty line and then local dependencies."

G2. NatSpec is wrong:

DeleGator/src/enforcers/DeployedEnforcer.sol:L29-L30

```
* @param _terms This is packed bytes where the first 20 bytes are where the contract should be deployed, the next 20 bytes are
* the deployer factory, and the remaining bytes are the bytecode to deploy.
```

The factory address is set as `immutable` in the constructor; it's not part of the `terms`.

G3. The variable name `bytecode_` in `beforeHook` and `getTermsInfo` is wrong/misleading:

DeleGator/src/enforcers/DeployedEnforcer.sol:L33-L40

```
(address contractAddress_, bytes memory bytecode_) = getTermsInfo(_terms);
// check if this contract has been deployed yet
if (contractAddress_.code.length > 0) {
    // if it has been deployed, then we don't need to do anything
    return;
}

bytes memory result_ = Address.functionCall(factoryAddress, bytecode_);
```

This is not bytecode; it's the entire calldata for the factory contract. A name like `factoryCalldata_` would be more accurate.

`ERC20AllowanceEnforcer`

H1. The `sender` and `delegationHash` should be indexed in the event `IncreasedSpendMap`:

DeleGator/src/enforcers/ERC20AllowanceEnforcer.sol:L19

```
event IncreasedSpendMap(address sender, bytes32 delegationHash, uint256 limit, uint256 spent);
```

H2. `address(uint160(bytes20(...)))` can be simplified to `address(bytes20(...))`:

DeleGator/src/enforcers/ERC20AllowanceEnforcer.sol:L71

```
allowedContract_ = address(uint160(bytes20(_terms[:20])));
```

The proposed structure `address(bytes20(...))` is used in `AllowedTargetsEnforcer`:

DeleGator/src/enforcers/AllowedTargetsEnforcer.sol:L53

```
allowedTargets_[j] = address(bytes20(_terms[i:i + 20]));
```

H3. `_terms.length` should be *exactly* 52, not `>=`.

DeleGator/src/enforcers/ERC20AllowanceEnforcer.sol:L69-L73

```
require(_terms.length >= 52, "ERC20AllowanceEnforcer:invalid-terms-length");

allowedContract_ = address(uint160(bytes20(_terms[:20])));
allowedMethod_ = IERC20.transfer.selector;
maxTokens_ = uint256(bytes32(_terms[20:]));
```

H4. Unnecessary return:

There is probably no good reason why `getTermsInfo` returns the constant (`IERC20.transfer.selector`) that's not part of the `terms`. The constant could be used directly in `beforeHook`.

DeleGator/src/enforcers/ERC20AllowanceEnforcer.sol:L64-L74

```
function getTermsInfo(bytes calldata _terms)
    public
    pure
    returns (address allowedContract_, bytes4 allowedMethod_, uint256 maxTokens_)
{
    require(_terms.length >= 52, "ERC20AllowanceEnforcer:invalid-terms-length");

    allowedContract_ = address(uint160(bytes20(_terms[:20])));
    allowedMethod_ = IERC20.transfer.selector;
    maxTokens_ = uint256(bytes32(_terms[20:]));
}
```

H5. Possible Panic Exception:

In `beforeHook`, if the length of `action.data` is less than 68, a Panic exception will be thrown in line 46 or 49, in the expression `_action.data[0:4]` or `_action.data[36:68]`.

DeleGator/src/enforcers/ERC20AllowanceEnforcer.sol:L46

```
bytes4 targetSig_ = bytes4(_action.data[0:4]);
```

DeleGator/src/enforcers/ERC20AllowanceEnforcer.sol:L49

```
uint256 sending_ = uint256(bytes32(_action.data[36:68]));
```

This doesn't cause any inherent risk, but according to the Solidity documentation: "Properly functioning code should never create a Panic". Hence, the function should first make sure that `_action.data.length` is exactly 68 and revert otherwise.

`ERC20BalanceGteEnforcer`

I1. Missing comment line:

```
    ////////////////////////////// State //////////////////////////////
```

DeleGator/src/enforcers/ERC20BalanceGteEnforcer.sol:L14-L19

```
contract ERC20BalanceGteEnforcer is CaveatEnforcer {
    mapping(address delegationManager => mapping(address delegator => mapping(address token => uint256 balance))) public
        balanceCache;
    mapping(address delegationManager => mapping(bytes32 delegationHash => bool lock)) public isLocked;

    ////////////////////////////// Public Methods //////////////////////////////
```

I2. Uncommon and inconsistent order of arguments in `terms`:

The amount comes first, then the token address.

DeleGator/src/enforcers/ERC20BalanceGteEnforcer.sol:L61

```
(uint256 amount_, address token_) = getTermsInfo(_terms);
```

It's more natural the other way around, which is also the order in `ERC20AllowanceEnforcer`.

DeleGator/src/enforcers/ERC20AllowanceEnforcer.sol:L42

```
(address allowedContract_, bytes4 allowedMethod_, uint256 limit_) = getTermsInfo(_terms);
```

`IncrementalIdEnforcer`

J1. The `delegator` should be indexed in the event `RevokedId`:

DeleGator/src/enforcers/IncrementalIdEnforcer.sol:L19

```
event RevokedId(address indexed delegationManager, address delegator, uint256 id);
```

`LimitedCallsEnforcer`

K1. `uint256(bytes32(_terms[:32]))` can be simplified to `uint256(bytes32(_terms))`:

Since the length of `terms` is exactly 32 bytes, the expression `uint256(bytes32(_terms[:32]))` can be simplified to `uint256(bytes32(_terms))`

DeleGator/src/enforcers/LimitedCallsEnforcer.sol:L52

```
limit_ = uint256(bytes32(_terms[:32]));
```

(cf. `IncrementalIdEnforcer` and `ValueLteEnforcer`).

`NonceEnforcer`

L1. The `delegator` should be indexed in event `UsedNonce`:

DeleGator/src/enforcers/NonceEnforcer.sol:L20

```
event UsedNonce(address indexed sender, address delegator, uint256 nonce);
```

L2. `uint256(bytes32(_terms[:32]))` can be simplified to `uint256(bytes32(_terms))`:

Since the length of `terms` is exactly 32 bytes, the expression `uint256(bytes32(_terms[:32]))` can be simplified to `uint256(bytes32(_terms))`

DeleGator/src/enforcers/NonceEnforcer.sol:L52

```
nonce_ = uint256(bytes32(_terms[:32]));
```

(cf. `IncrementalIdEnforcer` and `ValueLteEnforcer`).

L3. OpenZeppelin `BitMap` could be used:

Assuming the nonces (per delegation manager and delegator) are generally sequential, this enforcer could be made more efficient with an OpenZeppelin `BitMap`.

5.22 Miscellaneous Minor Points Affecting Several Contracts Minor

A. `public` functions that are not used internally should be `external` instead.

Exception: When a `public` function is overridden, it has to remain `public`, even if it's not used internally.

B. There is no consistent pattern to determine which functions in a contract are marked as `virtual` and which are not.

We recommend making deliberate and consistent decisions regarding the use of virtual functions.

Appendix 1 - Files in Scope

This audit covered the following files:

File	SHA-1 hash
DeleGator/src/DeleGatorCore.sol	`355940e4b551cfca4f56166dd5ec5bbd83868c4f`
DeleGator/src/DelegationManager.sol	`a823c7ffd265e8b0c7ce8f59a193d8aefa5c8297`
DeleGator/src/HybridDeleGator.sol	`8c16bfacaae3be310162aa0475fc4a0fb60bf75f`
DeleGator/src/MultiSigDeleGator.sol	`f2cd635d32cc4615ce215cebbf7c01d00b059b2f`
DeleGator/src/enforcers/AllowedCalldataEnforcer.sol	`7c330dddb9d297e065fea23fa2bc476f96054737`
DeleGator/src/enforcers/AllowedMethodsEnforcer.sol	`0419b86b7df33c3a569a1166cada93c909f16cb3`
DeleGator/src/enforcers/AllowedTargetsEnforcer.sol	`435d7c522aff0165bde509898a8ca94b48bca1c9`
DeleGator/src/enforcers/BlockNumberEnforcer.sol	`8f2e9373b01f990fd8b9576fd6256c8c2b15fc18`
DeleGator/src/enforcers/CaveatEnforcer.sol	`044e85e3124e268543417ebbb3b7ec145ec56d36`
DeleGator/src/enforcers/DeployedEnforcer.sol	`5aab86ea9648e0a9f58bd59ec57521378e8f3267`
DeleGator/src/enforcers/ERC20AllowanceEnforcer.sol	`c5a1a0ba5b802048723a7f8208572c18f737778b`
DeleGator/src/enforcers/ERC20BalanceGteEnforcer.sol	`86df5f232a948975246192698cf4b71247ea85d6`
DeleGator/src/enforcers/ERC20PaidEnforcer.sol	`6976196731a3b533ce5ea931982b7fe181763e1b`
DeleGator/src/enforcers/IncrementalIdEnforcer.sol	`b6b0ffc090cd5bdd661e6bd6a7fba1fb8826653d`
DeleGator/src/enforcers/LimitedCallsEnforcer.sol	`2dca64c6d30cd6a67d3c27df92563ad5833f4b2b`
DeleGator/src/enforcers/NonceEnforcer.sol	`6b66b1880600c1f93f941a63767722d8ec79510a`
DeleGator/src/enforcers/TimestampEnforcer.sol	`aa90a79972eeb5951f35fbc4183a4171ba3bbe39`
DeleGator/src/enforcers/ValueLteEnforcer.sol	`3bfccd9ea7d10422cc2daa16bb06ec31861a6d1a`
DeleGator/src/external/Daimo/P256.sol	`38e862045cedcbe4acc3cc02dc40eeec79a07d4e`
DeleGator/src/external/Daimo/P256Verifier.sol	`387c6d2ccd7e8d1955305222000985aaeb873eba`
DeleGator/src/external/Daimo/WebAuthn.sol	`5ff5826400f08809a61b78f324cd5b9243634228`
DeleGator/src/external/Daimo/utils/Base64URL.sol	`7625a76d3b4e9a9793df7585a0937c49815d733f`
DeleGator/src/interfaces/ICaveatEnforcer.sol	`bd79b3bc8df68fbdc78279c7b7a937832904d562`
DeleGator/src/interfaces/IDeleGatorCore.sol	`a6bd7fff8b1b7637ff993c9220476faa728c0161`
DeleGator/src/interfaces/IDeleGatorCoreFull.sol	`83b7e726a487ebfbabc56c2516e4b92ba8d69286`
DeleGator/src/interfaces/IDelegationManager.sol	`b66331b43126b4db9ff16ab8dda75e02a66008f7`
DeleGator/src/interfaces/IDelegationManagerMinimal.sol	`a8ebf2aae1eccda577e1aab4476b7841d3a1c494`
DeleGator/src/interfaces/IERC173.sol	`9cb693d0e5c26c7510b2c2fd0b8d38cbbc71653c`
DeleGator/src/libraries/ERC1271Lib.sol	`c649d9fe3edda0f3f44ace0c3994a14592cd741e`
DeleGator/src/libraries/ERC712Lib.sol	`800cf10a318c69b47ac55786fdc0115925377428`
DeleGator/src/libraries/EncoderLib.sol	`60b063d7f9604e026bd9e805ab44ca4d783808a7`
DeleGator/src/libraries/ExecutionLib.sol	`8d0dfa86200e1a81be924d98d3f92936dc731158`
DeleGator/src/libraries/P256VerifierLib.sol	`d8978372a4a843d3ec95eda340595f7fa7770cc6`
DeleGator/src/utils/SimpleFactory.sol	`4f27ba2529e66f56725989f183da1940b9bb6d1a`
DeleGator/src/utils/Typehashes.sol	`ac6cbe084f22220f1b2695bf0598c1c55a7b7e83`
DeleGator/src/utils/Types.sol	`e324399f6b44ff78f77f398767a338cbf092ca78`

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.