# Delegation Framework Audit Report

Prepared by Cyfrin

Version 2.0

**Lead Auditors**

0kage

Al-qaqa

March 18, 2025

## Contents

**1 About Cyfrin** **2**

**2 Disclaimer** **2**

**3 Risk Classification** **2**

**4 Protocol Summary** **2**

**5 Audit Scope** **3**

**6 Executive Summary** **4**

**7 Findings** **7**

7.1 High Risk . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 7

7.1.1 EntryPoint not included in user operation hash creates the possibility of Replay Attacks . . . 7

7.2 Medium Risk . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 10

7.2.1 Transfer Amount enforcer for ERC20 and Native transfers increase spend limit without checking actual transfers . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 10

7.2.2 Gas griefing via duplicate entries in `Allowed` class of enforcers . . . . . . . . . . . . . . . . 13

7.3 Low Risk . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 17

7.3.1 `EIP7702StatelessDeleGator` violates `EIP4337` signature validation standards . . . . . . . . 17

7.3.2 `AllowedCalldataEnforcer` cannot authenticate empty calldata preventing `receive()` function calls . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 17

7.3.3 NFT safe transfers will revert using `ERC721TransferEnforcer` . . . . . . . . . . . . . . . . . 18

7.3.4 Parameter mismatch in `IdEnforcer::beforeHook()` event emission . . . . . . . . . . . . . . 18

7.3.5 Inconsistent timestamp range validation in `TimestampEnforcer` . . . . . . . . . . . . . . . . 19

7.4 Informational . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 20

7.4.1 DelegationManager is incompatible with smart contract wallets with Approved hashes . . . . 20

7.4.2 `NotSelf()` error declaration is unused . . . . . . . . . . . . . . . . . . . . . . . . . . . . 21

7.4.3 Missing zero length check in `AllowedMethodsEnforcer::getTermsInfo()` . . . . . . . . . . 21

7.4.4 Insufficient delegate address validation in `NativeTokenPaymentEnforcer` . . . . . . . . . . . 21

7.5 Gas Optimization . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 23

7.5.1 `WebAuthn::contains()` optimization . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 23

## 1 About Cyfrin

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

## 2 Disclaimer

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

## 3 Risk Classification

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

## 4 Protocol Summary

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

*Key components of the DeleGator system include:*

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

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

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

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

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

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

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

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

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

## 5 Audit Scope

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

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

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

Core Protocol

- EIP7702DeleGatorCore.sol
- EIP7702StatelessDeleGator.sol
- HybridDeleGator.sol
- MultiSigDeleGator.sol
- DeleGatorCore.sol
- DelegationManager.sol
- SimpleFactory.sol

Libraries

- ERC1271Lib.sol
- EncoderLib.sol
- P256SCLVerifierLib.sol
- P256VerifierLib.sol

Interfaces

- ICaveatEnforcer.sol
- IDelegationManager.sol
- IDeleGatorCore.sol
- IERC173.sol
- IERC7821.sol

Enforcers

- AllowedCalldataEnforcer.sol
- AllowedMethodsEnforcer.sol
- AllowedTargetsEnforcer.sol

- ArgsEqualityCheckEnforcer.sol
- BlockNumberEnforcer.sol
- CaveatEnforcer.sol
- DeployedEnforcer.sol
- ERC20BalanceGteEnforcer.sol
- ERC20TransferAmountEnforcer.sol
- ERC721BalanceGteEnforcer.sol
- ERC721TransferEnforcer.sol
- ERC1155BalanceGteEnforcer.sol
- IdEnforcer.sol
- LimitedCallsEnforcer.sol
- NativeBalanceGteEnforcer.sol
- NativeTokenPaymentEnforcer.sol
- NativeTokenTransferAmountEnforcer.sol
- NonceEnforcer.sol
- OwnershipTransferEnforcer.sol
- RedeemerEnforcer.sol
- TimestampEnforcer.sol
- ValueLteEnforcer.sol

Utilities

- Constants.sol
- Types.sol

Scripts

- DeployEIP7702StatelessDeleGator.s.sol

## 6 Executive Summary

Over the course of 11 days, the Cyfrin team conducted an audit on the Delegation Framework smart contracts provided by Metamask. In this period, a total of 13 issues were found.

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

The security audit focused on the core delegation mechanics, signature verification across various account types (EOA, WebAuthn/passkey, multisig), enforcement of delegation constraints, and the overall security posture of the protocol. The review encompassed all key components including smart contract accounts, the delegation manager, policy enforcement modules (caveats), and their integration with ERC-4337.

The audit revealed a generally well-engineered codebase with strong security practices, though several significant issues were identified that require attention before deployment to production. Most notably, a high-risk vulnerability relating to potential replay attacks across different entry points was discovered. **All major issues found during the audit are successfully resolved.**

The DeleGator protocol demonstrates strong security practices including:

- Comprehensive test coverage with extensive unit and integration tests
- Clear separation of concerns between delegation, verification, and execution
- Robust signature validation across multiple signature schemes
- Thorough permission management with hierarchical delegation chains
- Well-structured caveat system for implementing fine-grained access controls

The test suite is extensive and covers most edge cases, especially around delegation chain validation and signature verification.

**Summary**

| | |
|---|---|
| Project Name | Delegation Framework |
| Repository | delegation-framework |
| Commit | d522a38b0b0f... |
| Audit Timeline | Feb 12th - Feb 26th, 2025 |
| Methods | Manual Review |

**Issues Found**

| | |
|---|---|
| Critical Risk | 0 |
| High Risk | 1 |
| Medium Risk | 2 |
| Low Risk | 5 |
| Informational | 4 |
| Gas Optimizations | 1 |
| Total Issues | 13 |

**Summary of Findings**

| | |
|---|---|
| [H-1] EntryPoint not included in user operation hash creates the possibility of Replay Attacks | Resolved |
| [M-1] Transfer Amount enforcer for ERC20 and Native transfers increase spend limit without checking actual transfers | Resolved |
| [M-2] Gas griefing via duplicate entries in `Allowed` class of enforcers | Acknowledged |
| [L-1] `EIP7702StatelessDeleGator` violates `EIP4337` signature validation standards | Resolved |
| [L-2] `AllowedCalldataEnforcer` cannot authenticate empty calldata preventing `receive()` function calls | Resolved |
| [L-3] NFT safe transfers will revert using `ERC721TransferEnforcer` | Resolved |
| [L-4] Parameter mismatch in `IdEnforcer::beforeHook()` event emission | Resolved |

| | |
|---|---|
| [L-5] Inconsistent timestamp range validation in `TimestampEnforcer` | Acknowledged |
| [I-1] DelegationManager is incompatible with smart contract wallets with Approved hashes | Resolved |
| [I-2] `NotSelf()` error declaration is unused | Acknowledged |
| [I-3] Missing zero length check in `AllowedMethodsEnforcer::getTermsInfo()` | Resolved |
| [I-4] Insufficient delegate address validation in `NativeTokenPaymentEnforcer` | Acknowledged |
| [G-1] `WebAuthn::contains()` optimization | Acknowledged |

# 7 Findings

## 7.1 High Risk

### 7.1.1 EntryPoint not included in user operation hash creates the possibility of Replay Attacks

**Description:** According to `EIP-4337`, the user operation hash should be constructed in a way to protect from replay attacks either on the same chain or different chains.

eip-4337#useroperation

> To prevent replay attacks (both cross-chain and multiple `EntryPoint` implementations), the `signature` should depend on `chainid` and the `EntryPoint` address
>
> The `userOpHash` is a hash over the userOp (except signature), entryPoint and chainId.

In the current `DeleGatorCore` and `EIP7702DeleGatorCore` implementations , the user operation hash does not include the `entryPoint` address.

```
    function validateUserOp(
        PackedUserOperation calldata _userOp,
>>      bytes32, //@audit ignores UserOpHash from the Entry Point
        uint256 _missingAccountFunds
    ) ... {
>>      validationData_ = _validateUserOpSignature(_userOp,
↪   getPackedUserOperationTypedDataHash(_userOp));
        _payPrefund(_missingAccountFunds);
    }
// ------------------
    function getPackedUserOperationTypedDataHash(PackedUserOperation calldata _userOp) public view
    ↪   returns (bytes32) {
        return MessageHashUtils.toTypedDataHash(_domainSeparatorV4(),
        ↪   getPackedUserOperationHash(_userOp));
    }
// ------------------
    function getPackedUserOperationHash(PackedUserOperation calldata _userOp) public pure returns
    ↪   (bytes32) {
        return keccak256(
            abi.encode(
                PACKED_USER_OP_TYPEHASH,
                _userOp.sender,
                _userOp.nonce,
                keccak256(_userOp.initCode),
                keccak256(_userOp.callData),
                _userOp.accountGasLimits,
                _userOp.preVerificationGas,
                _userOp.gasFees,
                keccak256(_userOp.paymasterAndData)
            )
        ); //@audeit does not include entry point address
    }
```

Note above that the `EntryPoint` address is not included in the hash generated via `getPackedUserOperationHash()`. The `_domainSeparatorV4()` will only include the `chainId` and `address(this)` but excludes the `EntryPoint` address.

**Impact:** Upgrading the delegator contract to include a new `EntryPoint` address opens the possibility of replay attacks.

**Proof of Concept:** Following POC shows the possibility of replaying previous native transfers when delegator contract is upgraded to a new `EntryPoint`.

```
contract EIP7702EntryPointReplayAttackTest is BaseTest {
```

```
using MessageHashUtils for bytes32;

constructor() {
    IMPLEMENTATION = Implementation.EIP7702Stateless;
    SIGNATURE_TYPE = SignatureType.EOA;
}

// New EntryPoint to upgrade to
EntryPoint newEntryPoint;
// Implementation with the new EntryPoint
EIP7702StatelessDeleGator newImpl;

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

    // Deploy a second EntryPoint
    newEntryPoint = new EntryPoint();
    vm.label(address(newEntryPoint), "New EntryPoint");

    // Deploy a new implementation connected to the new EntryPoint
    newImpl = new EIP7702StatelessDeleGator(delegationManager, newEntryPoint);
    vm.label(address(newImpl), "New EIP7702 StatelessDeleGator Impl");
}

function test_replayAttackAcrossEntryPoints() public {
    // 1. Create a UserOp that will be valid with the original EntryPoint
    address aliceDeleGatorAddr = address(users.alice.deleGator);

    // A simple operation to transfer ETH to Bob
    Execution memory execution = Execution({ target: users.bob.addr, value: 1 ether, callData:
    ↪  hex"" });

    // Create the UserOp with current EntryPoint
    bytes memory userOpCallData = abi.encodeWithSignature(EXECUTE_SINGULAR_SIGNATURE, execution);
    PackedUserOperation memory userOp = createUserOp(aliceDeleGatorAddr, userOpCallData);

    // Alice signs it with the current EntryPoint's context
    userOp.signature = signHash(users.alice, getPackedUserOperationTypedDataHash(userOp));

    // Bob's initial balance for verification
    uint256 bobInitialBalance = users.bob.addr.balance;

    // Execute the original UserOp through the first EntryPoint
    PackedUserOperation[] memory userOps = new PackedUserOperation[](1);
    userOps[0] = userOp;
    vm.prank(bundler);
    entryPoint.handleOps(userOps, bundler);

    // Verify first execution worked
    uint256 bobBalanceAfterExecution = users.bob.addr.balance;
    assertEq(bobBalanceAfterExecution, bobInitialBalance + 1 ether);

    // 2. Modify code storage
    // The code will be: 0xef0100 || address of new implementation
    vm.etch(aliceDeleGatorAddr, bytes.concat(hex"ef0100", abi.encodePacked(newImpl)));

    // Verify the implementation was updated
    assertEq(address(users.alice.deleGator.entryPoint()), address(newEntryPoint));

    // 3. Attempt to replay the original UserOp through the new EntryPoint
    vm.prank(bundler);
    newEntryPoint.handleOps(userOps, bundler);
```

```
        // 4. Verify if the attack succeeded - check if Bob received ETH again
        assertEq(users.bob.addr.balance, bobBalanceAfterExecution + 1 ether);

        console.log("Bob's initial balance was: %d", bobInitialBalance / 1 ether);
        console.log("Bob's balance after execution on old entry point was: %d",
        ↪  bobBalanceAfterExecution / 1 ether);
        console.log("Bob's balance after replaying user op on new entry point: %d",
        ↪  users.bob.addr.balance / 1 ether);
    }
}
```

**Recommended Mitigation:** Consider including `EntryPoint` address in the hashing logic of `getPackedUserOperationHash` of both `DeleGatorCore` and `EIP7702DeleGatorCore`

**Metamask:** Fixed in commit 1f91637

**Cyfrin:** Resolved. `EntryPoint` is now included in the hashing logic.

## 7.2 Medium Risk

### 7.2.1 Transfer Amount enforcer for ERC20 and Native transfers increase spend limit without checking actual transfers

**Description:** Failed token/native transfers can potentially deplete a delegation's spending allowance when used with the `EXECTYPE_TRY` execution mode.

The issue occurs because the enforcer tracks spending limits by incrementing a counter in its `beforeHook`, before the actual token transfer occurs. In the `EXECTYPE_TRY` mode, if the token transfer fails, the execution continues without reverting, but the limit is still increased.

```
function _validateAndIncrease(
    bytes calldata _terms,
    bytes calldata _executionCallData,
    bytes32 _delegationHash
)
    internal
    returns (uint256 limit_, uint256 spent_)
{
    // ... validation code ...

    //@audit This line increases the spent amount BEFORE the actual transfer happens
    spent_ = spentMap[msg.sender][_delegationHash] += uint256(bytes32(callData_[36:68]));
    require(spent_ <= limit_, "ERC20TransferAmountEnforcer:allowance-exceeded");
}
```

This means a malicious delegate could repeatedly attempt transfers that are designed to fail, draining the allowance without actually transferring any tokens.

**Impact:** This vulnerability allows an attacker to potentially exhaust a delegator's entire token transfer allowance without actually transferring any tokens

**Proof of Concept:**

```
function test_transferFailsButSpentLimitIncreases() public {
        // Create a delegation from Alice to Bob with spending limits
        Caveat[] memory caveats = new Caveat[](3);

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

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

        // ERC20 Transfer Amount Enforcer - limit to TRANSFER_LIMIT tokens
        caveats[2] = Caveat({
            enforcer: address(transferAmountEnforcer),
            terms: abi.encodePacked(address(mockToken), uint256(TRANSFER_LIMIT)),
            args: hex""
        });

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

        // Sign the delegation
        delegation = signDelegation(users.alice, delegation);

        // First, verify the initial spent amount is 0
        bytes32 delegationHash = EncoderLib._getDelegationHash(delegation);
        uint256 initialSpent = transferAmountEnforcer.spentMap(address(delegationManager),
            ↪  delegationHash);
        assertEq(initialSpent, 0, "Initial spent should be 0");

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

        // Amount to transfer
        uint256 amountToTransfer = 500 ether;

        // Create the mode for try execution
        ModeCode tryExecuteMode = ModeLib.encode(CALLTYPE_SINGLE, EXECTYPE_TRY, MODE_DEFAULT,
            ↪  ModePayload.wrap(bytes22(0x00)));

        // First test successful transfer
        {
            // Make sure token transfers will succeed
            mockToken.setHaltTransfer(false);

            // Prepare transfer execution
            Execution memory execution = Execution({
                target: address(mockToken),
                value: 0,
                callData: abi.encodeWithSelector(
                    IERC20.transfer.selector,
                    address(users.bob.addr), // Transfer to Bob's EOA
                    amountToTransfer
                )
            });

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

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

            // Check spent map was updated
            uint256 spentAfterSuccess = transferAmountEnforcer.spentMap(address(delegationManager),
                ↪  delegationHash);
            console.log("Spent amount after successful transfer:", spentAfterSuccess / 1e18);
```

```
            assertEq(spentAfterSuccess, amountToTransfer, "Spent amount should be updated after
            ↪  successful transfer");

            // Verify the transfer actually occurred
            assertEq(aliceBalanceAfterSuccess, aliceInitialBalance - amountToTransfer);
            assertEq(bobBalanceAfterSuccess, bobInitialBalance + amountToTransfer);
        }

        // Now test failing transfer
        {
            // Make token transfers fail
            mockToken.setHaltTransfer(true);

            // Prepare failing transfer execution
            Execution memory execution = Execution({
                target: address(mockToken),
                value: 0,
                callData: abi.encodeWithSelector(
                    IERC20.transfer.selector,
                    address(users.bob.addr), // Transfer to Bob's EOA
                    amountToTransfer
                )
            });

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

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

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

            // THE KEY TEST: The spent amount increased even though the transfer failed!
            assertEq(spentAfterFailure, amountToTransfer * 2, "Spent amount should increase even with
            ↪  failed transfer");

            // Verify tokens weren't actually transferred
            assertEq(aliceBalanceAfterFailure, aliceInitialBalance - amountToTransfer);
            assertEq(bobBalanceAfterFailure, bobInitialBalance + amountToTransfer);
        }
    }
```

**Recommended Mitigation:** Consider one of the following options:

1. Implement a post check in afterHook() with following steps
    - track the initial balance in beforeHook

    - track the actual balance in afterHook
    - only update spend limit based on actual - initial -> in the afterHook

2. Alternatively, enforce `TransferAmountEnforcers` to be of execution type `EXECTYPE_DEFAULT` only,

**Metamask:** Fixed in commit cdd39c6

**Cyfrin:** Resolved. Restricted execution type to only `EXECTYPE_DEFAULT`

### 7.2.2 Gas griefing via duplicate entries in `Allowed` class of enforcers

**Description:** Multiple enforcer contracts (`AllowedMethodsEnforcer` and `AllowedTargetsEnforcer`) don't validate the uniqueness of entries in their terms data, allowing malicious users to intentionally create delegations with excessive duplicates, dramatically increasing gas costs during validation.

`AllowedMethodsEnforcer`: Allows duplicate method selectors (4 bytes each) `AllowedTargetsEnforcer`: Allows duplicate target addresses (20 bytes each)

None of these contracts prevent or detect duplicates in their terms data. This allows an attacker to artificially inflate gas costs by including the same entries multiple times, resulting in expensive linear search operations during validation.

```
function getTermsInfo(bytes calldata _terms) public pure returns (bytes4[] memory allowedMethods_) {
    uint256 j = 0;
    uint256 termsLength_ = _terms.length;
    require(termsLength_ % 4 == 0, "AllowedMethodsEnforcer:invalid-terms-length");
    allowedMethods_ = new bytes4[](termsLength_ / 4);
    for (uint256 i = 0; i < termsLength_; i += 4) {
        allowedMethods_[j] = bytes4(_terms[i:i + 4]);
        j++;
    }
}

// In beforeHook:
for (uint256 i = 0; i < allowedSignaturesLength_; ++i) {
    if (targetSig_ == allowedSignatures_[i]) { //@audit linear search can be expensive
        return;
    }
}
```

**Impact:** As demonstrated in the test case for `AllowedMethodsEnforcer`, including 100 duplicate method signatures increases gas consumption from 50,881 to 155,417 (a difference of 104,536 gas). This represents a ~3x increase in gas consumption with just 100 duplicates. A malicious actor can use this to make execution expensive for a delegator.

**Proof of Concept:** Run the following test

```
 function test_AllowedMethods_DuplicateMethodsGriefing() public {
        // Create terms with a high number of duplicated methods to increase gas costs
        bytes memory terms = createDuplicateMethodsTerms(INCREMENT_SELECTOR, 100);

        // Create execution to increment counter
        Execution memory execution =
            Execution({ target: address(aliceCounter), value: 0, callData:
            ↪  abi.encodeWithSelector(INCREMENT_SELECTOR) });

        // Create delegation with allowed methods caveat
        Caveat[] memory caveats = new Caveat[](1);
        caveats[0] = Caveat({ enforcer: address(allowedMethodsEnforcer), terms: terms, args: "" });

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

        delegation = signDelegation(users.alice, delegation);

        // Measure gas usage with many duplicate methods
        Delegation[] memory delegations = new Delegation[](1);
        delegations[0] = delegation;

        uint256 gasUsed = uint256(
            bytes32(
                gasReporter.measureGas(
                    address(users.bob.deleGator),
                    address(delegationManager),
                    abi.encodeWithSelector(
                        delegationManager.redeemDelegations.selector,
                        createPermissionContexts(delegation),
                        createModes(),
                        createExecutionCallDatas(execution)
                    )
                )
            )
        );

        console.log("Gas used with 100 duplicate methods:", gasUsed);

        // Now compare to normal case with just one method
        terms = abi.encodePacked(INCREMENT_SELECTOR);
        caveats[0].terms = terms;

        delegation.caveats = caveats;
        delegation = signDelegation(users.alice, delegation);

        delegations[0] = delegation;

        uint256 gasUsedNormal = uint256(
            bytes32(
                gasReporter.measureGas(
                    address(users.bob.deleGator),
                    address(delegationManager),
                    abi.encodeWithSelector(
                        delegationManager.redeemDelegations.selector,
                        createPermissionContexts(delegation),
                        createModes(),
                        createExecutionCallDatas(execution)
                    )
                )
            )
        );

        console.log("Gas used with 1 method:", gasUsedNormal);
        console.log("Gas diff:", gasUsed - gasUsedNormal);

        assertGt(gasUsed, gasUsedNormal, "Griefing with duplicate methods should use more gas");
    }
    function createDuplicateMethodsTerms(bytes4 selector, uint256 count) internal pure returns (bytes
    ↪  memory) {
        bytes memory terms = new bytes(count * 4);
        for (uint256 i = 0; i < count; i++) {
            bytes4 methodSig = selector;
            for (uint256 j = 0; j < 4; j++) {
                terms[i * 4 + j] = methodSig[j];
            }
        }
        return terms;
    }

    function createPermissionContexts(Delegation memory del) internal pure returns (bytes[] memory) {
        Delegation[] memory delegations = new Delegation[](1);
        delegations[0] = del;

        bytes[] memory permissionContexts = new bytes[](1);
        permissionContexts[0] = abi.encode(delegations);

        return permissionContexts;
    }

    function createExecutionCallDatas(Execution memory execution) internal pure returns (bytes[]
    ↪  memory) {
        bytes[] memory executionCallDatas = new bytes[](1);
        executionCallDatas[0] = ExecutionLib.encodeSingle(execution.target, execution.value,
        ↪  execution.callData);
        return executionCallDatas;
    }

    function createModes() internal view returns (ModeCode[] memory) {
        ModeCode[] memory modes = new ModeCode[](1);
        modes[0] = mode;
        return modes;
    }
```

**Recommended Mitigation:** Consider enforcing that entries in terms are strictly increasing, which naturally prevents duplicates. A validation as shown below will prevent duplicates in this case. Also, consider implementing a binary search on a sorted array viz-a-viz linear search.

```
function getTermsInfo(bytes calldata _terms) public pure returns (bytes4[] memory allowedMethods_) {
    uint256 termsLength_ = _terms.length;
    require(termsLength_ % 4 == 0, "AllowedMethodsEnforcer:invalid-terms-length");
    allowedMethods_ = new bytes4[](termsLength_ / 4);

    bytes4 previousSelector = bytes4(0);

    for (uint256 i = 0; i < termsLength_; i += 4) {
        bytes4 currentSelector = bytes4(_terms[i:i + 4]);

        // Ensure selectors are strictly increasing (prevents duplicates)
        require(uint32(currentSelector) > uint32(previousSelector),
                "AllowedMethodsEnforcer:selectors-must-be-strictly-increasing"); //@audit prevents
                ↪  duplicates

        allowedMethods_[i/4] = currentSelector;
        previousSelector = currentSelector;
    }
}
```

**Metamask:** Acknowledged. It is the responsibility of the redeemer to see the terms are setup in such a way to prevent griefing attacks.

**Cyfrin:** Acknowledged.

## 7.3 Low Risk

### 7.3.1 `EIP7702StatelessDeleGator` violates `EIP4337` signature validation standards

**Description:** The `EIP7702StatelessDeleGator` implementation doesn't properly adhere to the EIP4337 standard regarding signature validation behavior.

According to EIP4337, the `validateUserOp` method should:

1. Return SIG_VALIDATION_FAILED (without reverting) only for signature mismatch cases
2. Revert for any other errors (including invalid signature format, incorrect length)

The current implementation in `EIP7702StatelessDeleGator._isValidSignature()` returns `SIG_VALIDATION_FAILED` for both signature mismatches and invalid signature length, which violates the specification:

```
// EIP7702StatelessDeleGator.sol
    function _isValidSignature(bytes32 _hash, bytes calldata _signature) internal view override returns
    ↪   (bytes4) {
>>      if (_signature.length != SIGNATURE_LENGTH) return ERC1271Lib.SIG_VALIDATION_FAILED;

        if (ECDSA.recover(_hash, _signature) == address(this)) return ERC1271Lib.EIP1271_MAGIC_VALUE;

        return ERC1271Lib.SIG_VALIDATION_FAILED;
    }
```

This implementation is used by the `EntryPoint` when validating `UserOperations`, and incorrect handling can lead to inconsistencies with other EIP4337-compliant wallets.

**Impact:** Creates potential inconsistency with other EIP4337-compliant wallets

**Recommended Mitigation:** Consider removing the signature length check in `EIP7702StatelessDeleGator._isValidSignature()`. This will cause the function to rely on OpenZeppelin's `ECDSA.recover()` function to revert for invalid signature formats, consistent with the EIP4337 specification.

**Metamask** Fixed in b52cf04.

**Cyfrin** Resolved.

### 7.3.2 `AllowedCalldataEnforcer` cannot authenticate empty calldata preventing `receive()` function calls

**Description:** The `AllowedCalldataEnforcer` contract has a design flaw that prevents it from authenticating calls with empty calldata. This issue arises from a requirement check in the `getTermsInfo` function that enforces `_terms.length >= 33`:

```
// AllowedCalldataEnforcer.sol
    function getTermsInfo(bytes calldata _terms) public pure returns (uint256 dataStart_, bytes memory
    ↪   value_) {
>>      require(_terms.length >= 33, "AllowedCalldataEnforcer:invalid-terms-size");
        dataStart_ = uint256(bytes32(_terms[0:32]));
        value_ = _terms[32:];
    }
```

The first 32 bytes represent the starting offset in the calldata, and anything after that represents the expected value to match against. This design requires at least 1 byte for the value, which prevents the enforcer from handling empty calldata scenarios.

Ethereum contracts can receive ETH through functions with empty calldata, specifically:

- When calling a contract's `receive()` function, which requires empty calldata
- When making simple ETH transfers to contracts that implement `receive()`

**Impact:** Users cannot use `AllowedCalldataEnforcer` to authorize delegations that should only permit simple ETH transfers to contracts with receive(). Common use cases like depositing ETH to WETH (which uses the receive() function) cannot be properly enforced through this caveat

**Recommended Mitigation:** Consider modifying the `getTermsInfo` function to allow terms of exactly 32 bytes length, treating it as a special case where the value is empty:

```
function getTermsInfo(bytes calldata _terms) public pure returns (uint256 dataStart_, bytes memory
↪   value_) {
    require(_terms.length >= 32, "AllowedCalldataEnforcer:invalid-terms-size");
    dataStart_ = uint256(bytes32(_terms[0:32]));
    if (_terms.length == 32) {
        value_ = new bytes(0); // @audit Empty bytes for empty calldata
    } else {
        value_ = _terms[32:];
    }
}
```

**Metamask:** Addressed in commit db11bf7.

**Cyfrin:** Resolved. Added a new caveat `ExactCalldataEnforcer` to address the issue.

### 7.3.3 NFT safe transfers will revert using `ERC721TransferEnforcer`

**Description:** The `ERC721TransferEnforcer` enforcer is designed to authorize NFT token transfers, but it currently restricts transfers to only use the `transferFrom` selector. The issue lies in this code section:

```
//ERC721TransferEnforcer.sol
bytes4 selector_ = bytes4(callData_[0:4]);

if (target_ != permittedContract_) {
    revert("ERC721TransferEnforcer:unauthorized-contract-target");
} else if (selector_ != IERC721.transferFrom.selector) {
    revert("ERC721TransferEnforcer:unauthorized-selector");
} else if (transferTokenId_ != permittedTokenId_) {
    revert("ERC721TransferEnforcer:unauthorized-token-id");
}
```

The ERC721 standard includes multiple transfer methods:

`transferFrom(address from, address to, uint256 tokenId)` `safeTransferFrom(address from, address to, uint256 tokenId)` `safeTransferFrom(address from, address to, uint256 tokenId, bytes data)`

The enforcer currently only supports the `transferFrom` method, which doesn't perform receiver capability checks. The `safeTransferFrom` methods are crucial for safely transferring NFTs to contracts, as they check whether the receiving contract supports the ERC721 standard through the onERC721Received callback.

**Impact:** Users are unable to utilize safer NFT transfer methods when using this enforcer.

**Recommended Mitigation:** Consider modifying the `ERC721TransferEnforcer` to accept all valid ERC721 transfer selectors.

**Metamask:** Fixed in 1a0ff2d.

**Cyfrin:** Resolved.

### 7.3.4 Parameter mismatch in `IdEnforcer::beforeHook()` event emission

**Description:** There's a parameter ordering mismatch in the IdEnforcer contract when emitting the `UsedId` event in the `beforeHook()` function. The event declaration and the actual emission use different parameter orders for the delegator and redeemer arguments.

The event is declared with parameters in this order:

```
//IdEnforcer.sol
    event UsedId(address indexed sender, address indexed delegator, address indexed redeemer, uint256
    ↪  id);
// ---------
    function beforeHook( ... ) ... {
        ...
>>      emit UsedId(msg.sender, _redeemer, _delegator, id_);
    }
```

**Impact:** Incorrect event data indexing.

**Recommended Mitigation:** Consider swapping the positions of `_redeemer` and `_delegator` in the event emission to align with the event declaration.

**Metamask:** Fixed in 5f8bea9

**Cyfrin:** Resolved.

### 7.3.5 Inconsistent timestamp range validation in `TimestampEnforcer`

**Description:** The `TimestampEnforcer` contract allows the creation of delegations with a time-based validity window. However, it lacks validation to ensure logical consistency of the time range.

Specifically, when both the "after" and "before" thresholds are specified, there is no check to ensure that `timestampBeforeThreshold_` is greater than `timestampAfterThreshold_`.

```
//TimeStampEnforcer.sol

function getTermsInfo(bytes calldata _terms)
    public
    pure
    returns (uint128 timestampAfterThreshold_, uint128 timestampBeforeThreshold_)
{
    require(_terms.length == 32, "TimestampEnforcer:invalid-terms-length");
    timestampBeforeThreshold_ = uint128(bytes16(_terms[16:]));
    timestampAfterThreshold_ = uint128(bytes16(_terms[:16]));
}
```

When both timestamps are non-zero, there is an independent check as follows

```
//TimeStampEnforcer.sol

require(block.timestamp > timestampAfterThreshold_, "TimestampEnforcer:early-delegation");
require(block.timestamp < timestampBeforeThreshold_, "TimestampEnforcer:expired-delegation");
//@audit no check on validity of the range
```

This creates a potential for inconsistent time ranges where the "after" threshold is greater than the "before" threshold, resulting in a permanently unusable delegation.

**Impact:** Delegations can be created that appear valid but can never be exercised because of impossible time constraints.

**Recommended Mitigation:** Consider adding a validation check in the `getTermsInfo` function to ensure that when both timestamp thresholds are non-zero, the "before" threshold is greater than the "after" threshold.

**Metamask:** Acknowledged. It is on the delegator to create the correct terms for the delegation.

**Cyfrin:** Acknowledged.

## 7.4 Informational

### 7.4.1 DelegationManager is incompatible with smart contract wallets with Approved hashes

**Description:** In the `DelegationManager` contract, there is a limitation that affects smart contract wallets that implement pre-approved hashes functionality, such as Safe (formerly Gnosis Safe) wallets. The current implementation rejects delegations with empty signatures for both EOA and smart contract wallets.

```
// DelegationManager.sol
    function redeemDelegations( ... ) ... {
        ...
        for (uint256 batchIndex_; batchIndex_ < batchSize_; ++batchIndex_) {
            ...
            if (delegations_.length == 0) { ... } else {
                ...
                for (uint256 delegationsIndex_; delegationsIndex_ < delegations_.length;
                ↪   ++delegationsIndex_) {
                    ...
>>                  if (delegation_.signature.length == 0) {
                        // Ensure that delegations without signatures revert
                        revert EmptySignature();
                    }

                    if (delegation_.delegator.code.length == 0) {
                        // Validate delegation if it's an EOA
                        ...
                    } else {
                        // Validate delegation if it's a contract
                        ...
>>                      bytes32 result_ =
↪   IERC1271(delegation_.delegator).isValidSignature(typedDataHash_, delegation_.signature);
                        ...
                    }
                }
```

This presents a compatibility issue with Safe wallets and similar smart contract wallets that use a pattern where empty signatures trigger checking for pre-approved hashes. In Safe's implementation:

SignatureVerifierMuxer.sol#L163-L165

```
    function isValidSignature(bytes32 _hash, bytes calldata signature) external view override returns
    ↪   (bytes4 magic) {
        (ISafe safe, address sender) = _getContext();

        // Check if the signature is for an `ISafeSignatureVerifier` and if it is valid for the domain.
        if (signature.length >= 4) {
            ...

            // Guard against short signatures that would cause abi.decode to revert.
            if (sigSelector == SAFE_SIGNATURE_MAGIC_VALUE && signature.length >= 68) { ... }
        }

        // domainVerifier doesn't exist or the signature is invalid for the domain - fall back to the
        ↪   default
>>      return defaultIsValidSignature(safe, _hash, signature);
    }
// -----------------
    function defaultIsValidSignature(ISafe safe, bytes32 _hash, bytes memory signature) internal view
    ↪   returns (bytes4 magic) {
        bytes memory messageData = EIP712.encodeMessageData( ... );
        bytes32 messageHash = keccak256(messageData);
>>      if (signature.length == 0) {
```

```
            // approved hashes
>>          require(safe.signedMessages(messageHash) != 0, "Hash not approved");
        } else {
            // threshold signatures
            safe.checkSignatures(address(0), messageHash, signature);
        }
        magic = ERC1271.isValidSignature.selector;
    }
```

Since DelegationManager rejects empty signatures before calling `isValidSignature`, it prevents Safe wallets from using their pre-approved hash mechanism for delegations.

**Impact:** This limitation prevents Safe wallets and similar smart contract wallets from using their gas-efficient pre-approved hash mechanism with delegations.

**Recommended Mitigation:** To enable compatibility with Safe wallets' pre-approved hash mechanism, consider applying the empty signature check only to EOAs. Alternatively, consider documenting that Safe wallets pre-approved hashes are not supported in the current delegation framework.

**Metamask:** Fixed in commit 155d20c.

**Cyfrin:** Resolved.

### 7.4.2 `NotSelf()` error declaration is unused

**Description:** The `DeleGatorCore` contract and `EIP7702DeleGatorCore` contract both define a custom error called `NotSelf()`, but this error is never actually thrown anywhere in either contract or their derived implementations.

```
  // DeleGatorCore and EIP7702DeleGatorcore
   /// @dev Error thrown when the caller is not this contract.
   error NotSelf();
```

**Recommended Mitigation:** Consider removing the error from these two contracts

**Metamask:** Acknowledged. Will keep this incase inheriting implementations wish to leverage in the future.

**Cyfrin:** Acknowledged.

### 7.4.3 Missing zero length check in `AllowedMethodsEnforcer::getTermsInfo()`

**Description:** The `AllowedMethodsEnforcer` contract's `getTermsInfo()` function correctly validates that the provided terms length is divisible by 4 (as each method selector is 4 bytes), but it fails to reject empty terms (length == 0). Empty terms would technically pass the modulo check since `0 % 4 == 0`, but would result in an empty array of allowed methods.

**Impact:** A delegation with empty terms would never allow any method to be called, as the enforcer would iterate through an empty array of allowed methods and then revert with "AllowedMethodsEnforcer:method-not-allowed" instead of properly rejecting the invalid terms with "AllowedMethodsEnforcer:invalid-terms-length"

**Recommended Mitigation:** Consider adding a specific check to ensure the terms length is greater than zero.

**Metamask:** Fixed in commit cb2d4d7.

**Cyfrin:** Resolved.

### 7.4.4 Insufficient delegate address validation in `NativeTokenPaymentEnforcer`

**Description:** The `NativeTokenPaymentEnforcer` contract's `afterAllHook()` function calls `delegationManager.redeemDelegations()` to process a payment using an allowance delegation. However, it doesn't properly validate that the delegate address in the allowance delegation is either the enforcer contract itself (`address(this)`) or the special `ANY_DELEGATE` address.

```
   // NativeTokenPaymentEnforcer.sol
    function afterAllHook( ... ) ... {
        ...
        Delegation[] memory allowanceDelegations_ = abi.decode(_args, (Delegation[]));

        ...
        // Attempt to redeem the delegation and make the payment
>>      delegationManager.redeemDelegations(permissionContexts_, encodedModes_, executionCallDatas_);

        ...
    }
```

This validation is important because the delegation will only be successfully redeemed if the caller is the specified delegate or if the delegate is set to `ANY_DELEGATE`. Without this check, the payment process might fail unexpectedly when the caller (the `NativeTokenPaymentEnforcer` contract) isn't authorized as the delegate.

**Impact:** Payment transactions may revert unexpectedly

**Recommended Mitigation:** Consider adding a check to ensure that the delegate address in the allowance delegation is either `address(this)` or the special `ANY_DELEGATE` address

**Metamask:** Acknowledged. Will revert in the `DelegationManager`.

**Cyfrin:** Acknowledged.

## 7.5 Gas Optimization

### 7.5.1 `WebAuthn::contains()` optimization

**Description:** In the `WebAuthn::contains()` function, the current implementation checks for out-of-bounds conditions inside the loop on each iteration. This is inefficient as the same check is performed repeatedly.

```
//WebAuthn.sol
function contains(string memory substr, string memory str, uint256 location) internal pure returns
↪   (bool) {
    bytes memory substrBytes = bytes(substr);
    bytes memory strBytes = bytes(str);

    uint256 substrLen = substrBytes.length;
    uint256 strLen = strBytes.length;

    for (uint256 i = 0; i < substrLen; i++) {
        if (location + i >= strLen) {
            return false;
        }
        ...
    }

    return true;
}
```

**Recommended Mitigation:** Consider performinga single bounds check before entering the loop. This checks if the substring would fit within the main string at the specified location.

**Metamask:** Acknowledged. Taken from SmoothCyptoLib - will keep this for consistency.

**Cyfrin:** Acknowledged.