Delegation Framework - Total Balance Change Enforcers Audit Report

Prepared by Cyfrin

Version 2.0

Lead Auditors

0kage

Chinmay Farkya

September 1, 2025

Contents

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

5 Audit Scope

Current audit scope included changes made to the following files:

- ERC1155TotalBalanceChangeEnforcer.sol
- ERC20TotalBalanceChangeEnforcer.sol
- ERC721TotalBalanceChangeEnforcer.sol
- NativeTokenTotalBalanceChangeEnforcer.sol
- verify-enforcer-contracts.sh
- DeployCaveatEnforcers.s.sol
- CaveatEnforcers.md

6 Executive Summary

Over the course of 3 days, the Cyfrin team conducted an audit on the Delegation Framework - Total Balance Change Enforcers smart contracts provided by Metamask. In this period, a total of 5 issues were found.

Summary

Project Name	Delegation Framework - Total Balance Change Enforcers
Repository	delegation-framework
Commit	3ddf4f720c6f...
Audit Timeline	Aug 6th - Aug 8th
Methods	Manual Review

Issues Found

Critical Risk	0
High Risk	1
Medium Risk	0
Low Risk	1
Informational	3
Gas Optimizations	0
Total Issues	5

Summary of Findings

[H-1] `TotalBalanceEnforcer` validation bypass when mixed with state modifying enforcers	Resolved
[L-1] Overly restrictive balance consistency check in `TotalBalanceChange` class of enforcers can cause potential DoS	Resolved
[I-1] Insufficient documentation of "TotalBalanceChangeEnforcer" can lead to incorrect use of enforcers	Resolved
[I-2] Misleading documentation in afterAllHook() natspec	Resolved
[I-3] Incorrect contract natspec for ERC1155TotalBalanceChangeEnforcer	Resolved

7 Findings

7.1 High Risk

7.1.1 `TotalBalanceEnforcer` validation bypass when mixed with state modifying enforcers

Description: Total Balance Change Enforcers fail to validate the final balance state when mixed with state-modifying enforcers (such as `NativeTokenPaymentEnforcer`) in delegation chains.

The early return bypass in `afterAllHook` causes total balance enforcers positioned after state-modifying enforcers to skip validation entirely, allowing balance constraints to be violated without detection.

The vulnerability occurs because Total Balance change enforcers clean up their shared tracker after the first validation, causing subsequent enforcers to hit an early return even when they should validate the modified final state.

All total balance change enforcers contain this vulnerable pattern:

```
function afterAllHook(...) public override {
    // ... setup code ...

    BalanceTracker memory balanceTracker_ = balanceTracker[hashKey_];

    // @audit this returns early
    if (balanceTracker_.expectedIncrease == 0 && balanceTracker_.expectedDecrease == 0) return;

    // ... validation logic that gets bypassed ...

    delete balanceTracker[hashKey_]; // Cleanup after validation
}
```

Consider a delegation chain with mixed enforcer types affecting the same recipient, as follows:

```
Alice: `TotalBalanceChangeEnforcer` (validates net effect)
Bob: `NativeTokenPaymentEnforcer` (modifies balance during afterAllHook)
Dave: `TotalBalanceChangeEnforcer` (should validate final state)
```

Flow would be as follows:

1. `beforeAllHook` phase: Alice and Dave's enforcers accumulate balance requirements
2. Primary transaction executes
3. `afterAllHook` phase:

- Alice's enforcer: Validates net balance change against accumulated requirements, cleans shared tracker
- Bob's enforcer: `NativeTokenPaymentEnforcer` transfers additional ETH via delegationManager.redeemDelegations(...)

```
// NativeTokenPaymentEnforcer.sol
function afterAllHook(...) public override {
    // ... setup ...

    uint256 balanceBefore_ = recipient_.balance;

    // @audit Transfers ETH during afterAll ook
    delegationManager.redeemDelegations(permissionContexts_, encodedModes_, executionCallDatas_);

    uint256 balanceAfter_ = recipient_.balance;
    require(balanceAfter_ >= balanceBefore_ + amount_,
    ↪  "NativeTokenPaymentEnforcer:payment-not-received");
}
```

- Dave's enforcer: Finds cleaned tracker (expectedIncrease == 0 && expectedDecrease == 0) → Early return, no validation

Dave's enforcer never validates the final balance state that includes Bob's payment transfer.

Impact: Developers expect each `TotalBalanceChangeEnforcer` to validate the final balance state, specially when the enforcer order is AFTER a state changing enforcer such as `NativeTokenPaymentEnforcer`. Early return causes enforcers positioned after state-modifying enforcers to be completely bypassed. This is a possible scenario specially in a batched execution environment where `TotalBalanceChangeEnforcer` can share state across multiple executions in a single batch.

Proof of Concept: Add the following test to the test suite:

```
// SPDX-License-Identifier: MIT AND Apache-2.0
pragma solidity 0.8.23;

import { Implementation, SignatureType } from "./utils/Types.t.sol";
import { BaseTest } from "./utils/BaseTest.t.sol";
import { Execution, Caveat, Delegation } from "../src/utils/Types.sol";
import { Counter } from "./utils/Counter.t.sol";
import { EncoderLib } from "../src/libraries/EncoderLib.sol";
import { MessageHashUtils } from "@openzeppelin/contracts/utils/cryptography/MessageHashUtils.sol";
import { NativeTokenTotalBalanceChangeEnforcer } from
↪   "../src/enforcers/NativeTokenTotalBalanceChangeEnforcer.sol";
import { NativeTokenTransferAmountEnforcer } from
↪   "../src/enforcers/NativeTokenTransferAmountEnforcer.sol";
import { NativeTokenPaymentEnforcer } from "../src/enforcers/NativeTokenPaymentEnforcer.sol";
import { ArgsEqualityCheckEnforcer } from "../src/enforcers/ArgsEqualityCheckEnforcer.sol";
import { IDelegationManager } from "../src/interfaces/IDelegationManager.sol";
import { EIP7702StatelessDeleGator } from "../src/EIP7702/EIP7702StatelessDeleGator.sol";
import "forge-std/Test.sol";

/**
 * @title Total Balance Enforcer Vulnerability Test - 3-Chain Scenario
 * @notice Demonstrates the vulnerability where TotalBalanceChangeEnforcer gets bypassed
 *         in a 3-delegation chain: Alice -> Bob -> Dave
 *         - Alice uses NativeTokenTotalBalanceChangeEnforcer with a recipient as Alice
 *         - Bob uses NativeTokenPaymentEnforcer that transfers 3 ETH from Alice to a different
 ↪  recipient
 *         - Dave uses NativeTokenTotalBalanceChangeEnforcer with same recipient as Alice's
 *         - Dave executes a transaction that decreases Alice's balance by 0.3 ETH
 *         After Alice's balance check, the balance tracker is cleaned, and even though
 *         payment was done via Bob's NativeTokenPaymentEnforcer, Dave's check is bypassed.
 *         afterAllHook moves from root to leaf, Dave's NativeTokenTotalBalanceChangeEnforcer
 *        is not aware of the payment made by Bob, leading to a successful execution
 */
contract TotalBalanceEnforcer3ChainVulnerabilityTest is BaseTest {
    using MessageHashUtils for bytes32;

    // Enforcers
    NativeTokenTotalBalanceChangeEnforcer public balanceEnforcer;
    NativeTokenPaymentEnforcer public paymentEnforcer;
    NativeTokenTransferAmountEnforcer public transferEnforcer;
    ArgsEqualityCheckEnforcer public argsEnforcer;

    // Test addresses
    address public sharedRecipient;
    address public bobPaymentRecipient;

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

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

    // Deploy enforcers
    balanceEnforcer = new NativeTokenTotalBalanceChangeEnforcer();
    argsEnforcer = new ArgsEqualityCheckEnforcer();
    transferEnforcer = new NativeTokenTransferAmountEnforcer();
    paymentEnforcer = new
    ↪  NativeTokenPaymentEnforcer(IDelegationManager(address(delegationManager)),
    ↪  address(argsEnforcer));

    // Set up test addresses
    sharedRecipient = address(users.alice.deleGator);
    bobPaymentRecipient = address(0x1337);

    // Fund accounts
    vm.deal(address(users.alice.deleGator), 10 ether);
    vm.deal(address(users.bob.deleGator), 10 ether);
    vm.deal(address(users.dave.deleGator), 10 ether);
    vm.deal(sharedRecipient, 5 ether);

    // Fund EntryPoint
    vm.prank(address(users.alice.deleGator));
    entryPoint.depositTo{ value: 1 ether }(address(users.alice.deleGator));

    vm.prank(address(users.bob.deleGator));
    entryPoint.depositTo{ value: 1 ether }(address(users.bob.deleGator));

    vm.prank(address(users.dave.deleGator));
    entryPoint.depositTo{ value: 1 ether }(address(users.dave.deleGator));
}

/**
 * @dev Main vulnerability demonstration with 3-delegation chain
 * This test should succeed if vulnerability exists, fail if vulnerability is fixed
 */
function test_totalBalanceChange_with_PaymentEnforcer() public {
    uint256 initialBalance = sharedRecipient.balance;

    // 1: Create Alice -> Bob delegation
    (Delegation memory aliceToBob, bytes32 aliceToBobHash) = _createAliceToBobDelegation();

    // 2: Create Bob -> Dave delegation with payment enforcement
    (Delegation memory bobToDave, bytes32 bobToDaveHash) =
    ↪  _createBobToDaveDelegationWithPayment(aliceToBobHash);

    // 3: Create Dave's delegation
    Delegation memory daveDelegation = _createDaveDelegation(bobToDaveHash);

    // 4: Create execution - transfer 0.3 ETH from Alice's balance
    Execution memory execution = Execution({ target: address(0xBEEF), value: 0.3 ether, callData:
    ↪  "" });

    // 5: Set up delegation chain
    Delegation[] memory delegationChain = new Delegation[](3);
    delegationChain[0] = daveDelegation;
    delegationChain[1] = bobToDave;
    delegationChain[2] = aliceToBob;

    uint256 bobInitialBalance = bobPaymentRecipient.balance;

    // 6: Execute the delegation chain
```

```
        invokeDelegation_UserOp(users.dave, delegationChain, execution);

        uint256 finalBalance = sharedRecipient.balance;
        uint256 bobFinalBalance = bobPaymentRecipient.balance;

        // 7: Verify the vulnerability
        uint256 totalDecrease = initialBalance - finalBalance;
        uint256 bobPayment = bobFinalBalance - bobInitialBalance;

        //  Alice loses 3.3 ETH total (0.3 ETH from execution + 3 ETH from Native Token Payment
        ↪   enforcer),
        // Bob gets 3 ETH
        // EXPECTED BEHAVIOR: transaction should revert before reaching here
        assertEq(totalDecrease, 3.3 ether, "Total decrease should be 3.3 ETH");
        assertEq(bobPayment, 3 ether, "Bob should receive 3 ETH payment");
    }

    function _createAliceToBobDelegation() internal returns (Delegation memory, bytes32) {
        bytes memory terms = abi.encodePacked(
            true, // enforceDecrease
            sharedRecipient,
            uint256(1 ether) // max 1 ETH decrease
        );

        Caveat[] memory caveats = new Caveat[](1);
        caveats[0] = Caveat({ enforcer: address(balanceEnforcer), terms: terms, args: "" });

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

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

        return (delegation, delegationHash);
    }

    function _createBobToDaveDelegationWithPayment(bytes32 aliceToBobHash) internal returns (Delegation
    ↪   memory, bytes32) {
        // Step 1: Create the Bob -> Dave delegation structure first (without payment allowance)
        bytes memory bobPaymentTerms = abi.encodePacked(bobPaymentRecipient, uint256(3 ether));

        Caveat[] memory bobCaveats = new Caveat[](1);
        bobCaveats[0] = Caveat({
            enforcer: address(paymentEnforcer),
            terms: bobPaymentTerms,
            args: "" // Will be filled with allowance delegation
         });

        Delegation memory bobToDave = Delegation({
            delegate: address(users.dave.deleGator),
            delegator: address(users.bob.deleGator),
            authority: aliceToBobHash,
            caveats: bobCaveats,
            salt: 0,
            signature: ""
        });
        // Step 2: Sign to get the delegation hash
        bobToDave = signDelegation(users.bob, bobToDave);
        bytes32 bobToDaveHash = EncoderLib._getDelegationHash(bobToDave);

        // Step 3: Create allowance delegation that uses THIS delegation hash
        Delegation memory paymentAllowance = _createPaymentAllowanceDelegation(bobToDaveHash,
        ↪   address(users.dave.deleGator));

        // Step 4: Now set the args with the properly configured allowance delegation
        Delegation[] memory allowanceDelegations = new Delegation[](1);
        allowanceDelegations[0] = paymentAllowance;
        bobToDave.caveats[0].args = abi.encode(allowanceDelegations);

        return (bobToDave, bobToDaveHash);
    }

    function _createPaymentAllowanceDelegation(bytes32 mainDelegationHash, address redeemer) internal
    ↪   returns (Delegation memory) {
        // ArgsEqualityCheckEnforcer terms: pre-populated with delegation hash + redeemer
        // This matches the pattern from the working tests
        bytes memory argsEnforcerTerms = abi.encodePacked(mainDelegationHash, redeemer);

        Caveat[] memory caveats = new Caveat[](2);

        // Args enforcer with pre-populated terms
        caveats[0] = Caveat({
            enforcer: address(argsEnforcer),
            terms: argsEnforcerTerms, // Pre-populated with delegation hash + redeemer
            args: "" // Will be populated dynamically by PaymentEnforcer during execution
         });

        // Transfer amount enforcer - allows up to 3 ETH transfer
        caveats[1] = Caveat({ enforcer: address(transferEnforcer), terms: abi.encode(uint256(3 ether)),
        ↪   args: "" });

        Delegation memory delegation = Delegation({
            delegate: address(paymentEnforcer),
            delegator: address(users.alice.deleGator), // Alice delegates payment authority from her
            ↪   balance
            authority: ROOT_AUTHORITY,
            caveats: caveats,
            salt: 1,
            signature: ""
        });

        return signDelegation(users.alice, delegation);
    }

    function _createDaveDelegation(bytes32 bobToDaveHash) internal returns (Delegation memory) {
        bytes memory daveTerms = abi.encodePacked(
            true, // enforceDecrease
            sharedRecipient,
            uint256(0.5 ether) // max 0.5 ETH decrease
        );

        Caveat[] memory daveCaveats = new Caveat[](1);
        daveCaveats[0] = Caveat({ enforcer: address(balanceEnforcer), terms: daveTerms, args: "" });

        Delegation memory delegation = Delegation({
            delegate: address(users.dave.deleGator),
            delegator: address(users.dave.deleGator),
            authority: bobToDaveHash,
            caveats: daveCaveats,
            salt: 0,
            signature: ""
        });

        return signDelegation(users.dave, delegation);
    }
}
```

Recommended Mitigation: Consider adding an `validationsRemaining` in `BalanceTracker` that counts the pending `TotalBalanceChangeEnforcer` caveats remaining.

```
struct BalanceTracker {
    uint256 balanceBefore;
    uint256 expectedIncrease;
    uint256 expectedDecrease;
    uint256 validationsRemaining; // Countdown to final validation
}

function beforeAllHook(...) {
    // ... existing logic ...
    balanceTracker_.validationsRemaining++;
}

function afterAllHook(...) {
    // ... setup ...

    balanceTracker_.validationsRemaining--;

    // Only validate and cleanup when all total balance change enforcers have been processed
    if (balanceTracker_.validationsRemaining == 0) {
        // Perform validation
        // ... validation logic ...
        delete balanceTracker[hashKey_];
    }
}
```

Metamask: Fixed in PR 144.

Cyfrin: Verified.

7.2 Low Risk

7.2.1 Overly restrictive balance consistency check in `TotalBalanceChange` class of enforcers can cause potential DoS

Description: `TotalBalanceChange` class of enforcers contain a defensive check in `beforeAll` hook that can cause DoS attacks in cross-execution batch scenarios and prevents legitimate balance-modifying enforcers from being used.

The issue stems from overly restrictive validation that assumes balance immutability during the `beforeAll` hook phase.

```
function beforeAllHook(...) public override {
    // ... setup code ...

    uint256 currentBalance_ = IERC20(token_).balanceOf(recipient_);
    if (balanceTracker_.expectedDecrease == 0 && balanceTracker_.expectedIncrease == 0) {
        balanceTracker_.balanceBefore = currentBalance_;
        emit TrackedBalance(msg.sender, recipient_, token_, currentBalance_);
    } else {
        // @audit Overly restrictive check can prevent legitimate enforcers
        require(balanceTracker_.balanceBefore == currentBalance_,
        ↪  "ERC20TotalBalanceChangeEnforcer:balance-changed");
    }
    // ...
}
```

The check above assumes that there would not be any enforcers that can change state in the `beforeAll` hook. Imagine a `prepaymentEnforcer` implemented in future that requires upfront token transfer before execution (eg. fees or collateral moved to an escrow account) - current balance validation will make the `TotalBalanceChange` enforcers incompatible with such enforcers.

Consider following batch execution scenario:

```
// Execution 1: Bob uses TotalBalanceChangeEnforcer for Alice
// Execution 2: Dave uses PrepaymentEnforcer + TotalBalanceChangeEnforcer for Alice

// Attack flow:
// 1. Exec1 TotalBalanceChangeEnforcer.beforeAllHook: Sets balanceBefore = Alice's current balance
// 2. Exec2 PrepaymentEnforcer.beforeAllHook: Collects 1 ETH from Alice → Alice balance changes
// 3. Exec2 TotalBalanceChangeEnforcer.beforeAllHook: require(old_balance == new_balance) → REVERT
// 4. Entire batch transaction fails
```

Impact: Overly restrictive validation can DoS entire batch transactions when specific enforcer combinations are at play.

Recommended Mitigation: Consider replacing the restrictive balance consistency check with simple overwriting. Overwriting would mean that the last `TotalBalanceChangeEnforcer` becomes the baseline for all `afterAllHook` operations.

Current balance check is not adding any value from a "security" standpoint but adds DoS and enforcer incompatibility risks.

```
function beforeAllHook(...) public override {
    // ... setup code ...

    uint256 currentBalance_ = IERC20(token_).balanceOf(recipient_);

    if (balanceTracker_.expectedDecrease == 0 && balanceTracker_.expectedIncrease == 0) {
        balanceTracker_.balanceBefore = currentBalance_;
        emit TrackedBalance(msg.sender, recipient_, token_, currentBalance_);
    } else {
        // @audit Update baseline to current balance instead of requiring equality
        balanceTracker_.balanceBefore = currentBalance_;
    }
    // ...
}
```

Metamask: Fixed in PR 144.

Cyfrin: Verified.

7.3 Informational

7.3.1 Insufficient documentation of "TotalBalanceChangeEnforcer" can lead to incorrect use of enforcers

Description: The current documentation for Total Balance Change Enforcers is misleading and lacks clear guidance on when to use each enforcer type. This confusion may lead developers to incorrectly use `TotalBalanceChangeEnforcer` for independent security constraints instead of `BalanceChangeEnforcer`, resulting in weakened security through incorrect constrains.

The documentation fails to clearly distinguish between two fundamentally different use cases:

Independent security constraints - multiple delegations each imposing limits *Coordinated multi-operation transactions* single complex transaction with multiple steps

Current documentation provides following example without sufficient context

```
Consider a delegation chain with 3 instances of ERC20TotalBalanceChangeEnforcer:

Enforcer 1: Expects an increase of at least 1000 tokens
Enforcer 2: Expects an increase of at least 200 tokens
Enforcer 3: Expects a decrease of at most 300 tokens

Accumulate expected changes: +1000 + 200 - 300 = +900
Validate that the final balance has increased by at least 900 tokens
```

The documentation doesn't explain:

- Why accumulation is appropriate here vs. taking the most restrictive constraint
- When this pattern should be used
- What happens to individual enforcer requirements (Enforcer 1's 1000 requirement)

Current documentation lacks guidance for developers to choose between enforcer types, leading to incorrect usage patterns.

Another important insight missing is the fact that `TotalBalanceChangeEnforcer` class of caveats can persist state across multiple executions in a batched execution environment. Since the `hashkey` does not include delegation hash, the same `TotalBalanceChangeEnforcer` can share state across multiple, unrelated execution call datas, as long as the token and recipient are the same.

Impact: Developers using wrong enforcer type leading to constraint violations and security bypasses.

For example, if developer intent is to create progressively stricter delegation chain:

```
Alice delegates to Bob: "Treasury can lose max 100 ETH"
Bob delegates to Dave: "Treasury can lose max 50 ETH" (more restrictive)
Expected behavior: Enforce 50 ETH limit (stricter wins)

Using TotalBalanceChangeEnforcer:
Alice: TotalBalanceChangeEnforcer (expectedDecrease = 100 ETH)
Bob: TotalBalanceChangeEnforcer (expectedDecrease = 50 ETH)
Accumulation: 100 + 50 = 150 ETH total decrease allowed
```

In the above example, delegation chain becomes MORE permissive instead of more restrictive.

Recommended Mitigation: Consider:

1. Consider updating documentation top provide clear context to developers on when to use total balance change enforcer v/s balance change enforcer
2. Consider adding documentation that clearly specifies that multiple executions in a `batch execution` environment can use the same instance of `TotalBalanceChangeEnforcer` contracts and hence share state of total balances

3. Consider renaming `TotalBalanceChangeEnforcer` to `MultiOperationBalanceEnforcer` for better clarity

Metamask: Fixed in PR 144

Cyfrin: Verified.

7.3.2 Misleading documentation in afterAllHook() natspec

Description: In `ERC20TotalBalanceChangeEnforcer :: afterAllHook()` , the natspec says "This function validates that the recipient's token balance has changed by at least the total expected amount", which is incorrect.

In case of the net effect being expected decrease, the actual balance change allowed is limited by a max decrease, in which case balance does not change by "at least an expected amount".

Similarly, natspec of `afterAllHook()` in these enforcers is unclear :

- ERC1155TotalBalanceChangeEnforcer
- ERC721TotalBalanceChangeEnforcer
- NativeTokenTotalBalanceChangeEnforcer

For example, in `ERC1155TotalBalanceChangeEnforcer`, it says "This function enforces that the recipient's ERC1155 token balance has changed by the expected amount". But this could be interpreted as "Balance should always change by the expected amount", while in case of a net expected decrease, a change thats lower than that of "expected maxDecrease" is also allowed.

Impact: Incorrect natspec can mislead developers into misunderstanding the design of these enforcers.

Recommended Mitigation: The natspec at all these places shall be changed to "This function validates that the recipient's token balance has changed within expected limits".

Metamask: Resolved in PR 144.

Cyfrin: Verified.

7.3.3 Incorrect contract natspec for ERC1155TotalBalanceChangeEnforcer

Description: The contract natspec for `ERC1155TotalBalanceChangeEnforcer` has several instances of incorrect descriptions about how the balance state is stored in the enforcer.

- "Tracks initial balance and accumulates expected increases and decreases per recipient/token pair within a redemption" => which is incorrect as the accumulated expected amounts are stored per recipient-token-tokenID combination.
- "State is shared between enforcers watching the same recipient/token pair" => Again state is shared only for the same recipient-token-tokenID combination.

Impact: These comments incorrectly describe the intended design of the `ERC1155TotalBalanceChangeEnforcer`. Including the tokenID here for deriving the storage keys is very important.

Recommended Mitigation: Consider modifying the natspec to correctly state that balance storage for ERC1155 enforcer is shared as per the `recipient-token-tokenID` combination.

Metamask: Fixed.

Cyfrin: Verified.