yAudit Sickle Uniswap and Curve integrations Review

Review Resources:

- Sickle documentation

Auditors:

- fedebianu
- watermelon

Table of Contents

Review Summary

Sickle

Sickle provides smart contract accounts that offer a wide range of integrations with popular DeFi applications across multiple EVM-compatible networks. Furthermore, Sickle provides services to automate common actions, such as liquidity rebalancing and reward compounding.

The contracts of the Sickle repository were reviewed over 12 days. Two auditors performed the code review between April 7 and 22, 2025. The repository was under active development during the review, but the review was limited to the latest commit 8ce412b25614ba8f6b01cd6406bc740926e6fadf for the Sickle repository.

All these changes are reflected in commit 74dfa3d33ef97b5b69cb91a21558dd53344ed108 of Sickle's public repo.

Scope

The scope of the review consisted of the following contracts at the specific commit:

```
contracts
├── ConnectorLens.sol
├── connectors
│   ├── curve
│   │   ├── CurveGaugeConnector.sol
│   │   ├── CurvePoolConnector.sol
│   │   └── CurveRegistry.sol
│   ├── UniswapV3Connector.sol
│   └── UniswapV4Connector.sol
├── libraries
│   ├── FeesLib.sol
│   ├── NftSettingsLib.sol
│   ├── PositionSettingsLib.sol
│   └── TransferLib.sol
```

```
├── NftSettingsRegistry.sol
└── PositionSettingsRegistry.sol
```

After the findings were presented to the Sickle team, fixes were made and included in several PRs.

This review is a code review to identify potential vulnerabilities in the code. The reviewers did not investigate security practices or operational security and assumed that privileged accounts could be trusted. The reviewers did not evaluate the security of the code relative to a standard or specification. The review may not have identified all potential attack vectors or areas of vulnerability.

yAudit and the auditors make no warranties regarding the security of the code and do not warrant that the code is free from defects. yAudit and the auditors do not represent nor imply to third parties that the code has been audited nor that the code is free from defects. By deploying or using the code, Sickle and users of the contracts agree to use the code at their own risk.

Code Evaluation Matrix

Category	Mark	Description
Access Control	Good	Correct access control is implemented within the reviewed contracts.
Mathematics	Average	Simple mathematical relations are used within the reviewed contracts. Incorrect decimal precision adjustment issues were identified when interacting with non-standard ERC20 tokens.
Complexity	Average	The reviewed contracts present a high degree of complexity, given that the system's architecture forces the used abstractions to integrate correctly with existing contracts, both internal and external to Sickle.
Libraries	Good	The reviewed contracts make use of the well-established Solmate and OpenZeppelin libraries.
Decentralization	Average	Interacting with the system may result in rather complex processes if not initiated via Sickle's UI; nonetheless, users are granted full power and control over how their Sickle interacts with integrated protocols and assets. Key changes to registries must be executed via a timelock, granting users a time buffer to take necessary actions and precautions if need be.
Code stability	Good	The codebase remained stable throughout the review.
Documentation	Average	Accurate NATSPEC documentation is present within the main entry points and key data structure definitions.
Monitoring	Average	The system correctly emits logs for key changes and interactions.
Testing and verification	Average	The reviewed contracts have been tested with unit and integration tests. The lack of unit tests for key, non-standard scenarios revealed higher severity issues.

Findings Explanation

Findings are broken down into sections by their respective impact:

- Critical, High, Medium, Low impact
    - These are findings that range from attacks that may cause loss of funds, impact control/ownership of the contracts, or cause any unintended consequences/actions that are outside the scope of the requirements.
- Gas savings

    - Findings that can improve the gas efficiency of the contracts.
- Informational
    - Findings including recommendations and best practices.

Critical Findings

None.

High Findings

1. High - `CurvePoolConnector.getPoolPrice()` provides incorrect input amount for pools containing tokens with non-standard decimals

`CurvePoolConnector.getPoolPrice()` calls into a given Curve pool to fetch the pool's current pricing of two assets in the pool. It achieves this by fetching a quote for the output amount of the swap's output token when providing `1e18` of the swap's input token.

Technical Details

The highlighted method pulls the price between the two assets by invoking the Curve pool's `get_dy` method, which expects to receive the amount of input token provided in a swap and returns the swap's output amount.

Hard-coding the input amount as `1e18` stands to create a significant error in the received quote in the case in which the input token does not have 18 decimals of fixed precision: this is the case for `USDC` and `USDT` on Ethereum which have six decimals of fixed precision or `GUSD` on Ethereum which only has two.

The curve pool will interpret `1e18` as a `10 ** (18 - token.decimals())` input amount, which will generate an error in the reported price's value and magnitude.

The method does this when requesting both the direct and inverse price for two assets:

- direct
- inverse

Impact

High. Querying a Curve pool's reported price can malfunction when dealing with tokens that have non-standard decimals, such as the most popular stablecoins.

Recommendation

Refactor the highlighted method to read the number of decimals used by the input token and use such value to determine the quote's input amount:

```
@@ -14,6 +14,10 @@ struct CurveSwapExtraData {
     int128 j;
 }

+interface IERC20Decimals {
+    function decimals() external view returns(uint256);
+}
+
@@ -69,18 +73,24 @@ contract CurvePoolConnector is ILiquidityConnector {
         uint256 baseTokenIndex,
         uint256 quoteTokenIndex
     ) external view override returns (uint256 price) {
+        address inputToken = ICurvePool(lpToken).coins(baseTokenIndex);
+        uint256 inputTokenDecimals = IERC20Decimals(inputToken).decimals();
+
         uint256 amountOut0 = ICurvePool(lpToken).get_dy(
             int128(uint128(baseTokenIndex)),
             int128(uint128(quoteTokenIndex)),
-            1e18
+            10 ** inputTokenDecimals
         );
         if (amountOut0 > 0) {
             price = amountOut0;
         } else {
+            inputToken = ICurvePool(lpToken).coins(quoteTokenIndex);
+            inputTokenDecimals = IERC20Decimals(inputToken).decimals();
+
             uint256 amountOut1 = ICurvePool(lpToken).get_dy(
                 int128(uint128(quoteTokenIndex)),
                 int128(uint128(baseTokenIndex)),
-                1e18
+                10 ** inputTokenDecimals
             );
```

```
            if (amountOut1 == 0) {
                revert InvalidPrice();
```

Developer Response

Fixed in commit 296d744.

2. High - `UniswapV4Connector.addLiquidity()` does not reclaim excess ETH

`UniswapV4Connector.addLiquidity()` is used by a Sickle to provide liquidity to the Uniswap V4 protocol: it can either mint a new position via `_mint()` or add to an existing one with `_increase_liquidity()`.

Technical Details

When providing liquidity to a Uniswap V4 pool, both internal methods use user input to specify the maximum amount of tokens to be provided for the pool in exchange for the requested amount of liquidity.

Because V4 pools now support native ETH as a pool's asset, users of the protocol are required to send it within calls to the `PositionManager.modifyLiquidities()` method. When doing so, if the sent amount of ETH were to exceed the effective amount used to mint the liquidity position, Sickle would not be able to recover it from the Uniswap pool.

Note that this issue is not present for ERC20 tokens, given that the exact amount is pulled from the user via a `transferFrom` call.

PoC

Create a foundry project with Uniswap v4 libraries:

```
forge init univ4-testing
cd univ4-testing
forge install https://github.com/Uniswap/v4-core
forge install https://github.com/Uniswap/v4-periphery
```

Create a new test file:

```
touch test/PoC.t.sol
```

Within the newly created file, insert the following content:

```
// SPDX-License-Identifier: UNLICENSED
pragma solidity ^0.8.13;

import {Test, console} from "forge-std/Test.sol";
import {Actions} from "v4-periphery/src/libraries/Actions.sol";
import {IStateView} from "v4-periphery/src/interfaces/IStateView.sol";
import {Currency} from "v4-core/types/Currency.sol";
import {PoolKey} from "v4-core/types/PoolKey.sol";
import {PoolId} from "v4-periphery/lib/v4-core/src/types/PoolId.sol";
import {PositionInfo, PositionInfoLibrary} from "v4-
periphery/src/libraries/PositionInfoLibrary.sol";

interface IPositionManager {
    function modifyLiquidities(bytes calldata unlockData, uint256 deadline)
external payable;
    function poolKeys(bytes25) external returns(PoolKey memory);
    function positionInfo(uint256) external returns(PositionInfo);
    function nextTokenId() external returns(uint256);
}

interface IAllowanceTransfer {
    function approve(address,address,uint160,uint48) external;
}

interface IERC20 {
    function approve(address,uint) external;
}

contract PoC is Test {
    IPositionManager posm;
    IStateView sv;
    IAllowanceTransfer permit2;
    address alice;

    bytes32 poolId;
    address usdc;

    function setUp() external {
        vm.createSelectFork(getChain(1).rpcUrl, 22273384);

        alice = makeAddr("alice");
```

```
        posm = IPositionManager(0xbD216513d74C8cf14cf4747E6AaA6420FF64ee9e);
        sv = IStateView(0x7fFE42C4a5DEeA5b0feC41C94C136Cf115597227);
        permit2 = IAllowanceTransfer(0x000000000022D473030F116dDEE9F6B43aC78BA3);
        poolId =
0x21c67e77068de97969ba93d4aab21826d33ca12bb9f565d8496e8fda8a82ca27; // eth-usdc
        usdc = 0xA0b86991c6218b36c1d19D4a2e9Eb0cE3606eB48;

        vm.deal(alice, 1 << 128);
        deal(usdc, alice, 1 << 128);

        assertGt(address(posm).code.length, 0);

        vm.prank(alice); IERC20(usdc).approve(address(permit2), type(uint).max);
        vm.prank(alice); permit2.approve(usdc, address(posm), type(uint160).max,
type(uint48).max);
    }

    function test_excess_eth_is_claimable_by_third_party() external {
        // Setup params to add liq
        int24 tickLower = -202290;
        int24 tickUpper = -202270;
        uint256 liq = 1e18;
        uint256 max0 = 1e21;
        uint256 max1 = 1e10;

        bytes memory actions = abi.encodePacked(
            uint8(Actions.MINT_POSITION),
            uint8(Actions.SETTLE_PAIR)
        );

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

        Currency currency0 = Currency.wrap(address(0));
        Currency currency1 = Currency.wrap(usdc);
        PoolKey memory key = posm.poolKeys(bytes25(poolId));

        params[0] = abi.encode(
            key,
            tickLower,
            tickUpper,
            liq,
            max0,
            max1,
```

```
            alice,
            "" // empty hookData
        );
        params[1] = abi.encode(currency0, currency1);

        uint256 valueToPass = max0; // alice sends max0 ETH to provide liq -
clearly overshoots amount needed

        uint256 aliceETHBal = alice.balance;

        vm.prank(alice);
        posm.modifyLiquidities{value: valueToPass}(
            abi.encode(actions, params),
            block.timestamp
        );

        uint256 aliceETHBalDelta = aliceETHBal - alice.balance;

        // Alice sent 1e21 wei
        assertEq(aliceETHBalDelta, valueToPass);

        //// Bob sweeps ETH
        address bob = makeAddr("bob");

        actions = abi.encodePacked(
            uint8(Actions.SWEEP)
        );
        params = new bytes[](1);
        params[0] = abi.encode(currency0, bob);

        // Bob has no initial balance
        assertEq(bob.balance, 0);

        vm.prank(bob);
        posm.modifyLiquidities(
            abi.encode(actions,params),
            block.timestamp
        );

        // Bob received ETH
        assertGt(bob.balance, 0);
        console.log("Bob gained %e", bob.balance);
```

```
    }
}
```

Execute the PoC:

```
forge t --mt test_excess_eth_is_claimable_by_third_party -vv
```

Impact

High. Excess ETH provided to Uniswap V4 when minting liquidity will not be reclaimed by the Sickle, enabling any account to claim it.

Recommendation

To correctly reclaim any excess ETH amount, calls to `PositionManager.modifyLiquidities()`, which provide ETH liquidity, should end with a `Action.SWEEP` operation to sweep any excess ETH.

Developer Response

Fixed in commit c6092a4.

Medium Findings

1. Medium - `UniswapV4Connector.addLiquidity()` breaks when using non-standard tokens like `USDT`

The `UniswapV4Connector.sol` contract doesn't implement the safe approval pattern when approving tokens for the `Permit2` contract. This can cause transactions to fail when using non-standard ERC20 tokens, such as USDT, which require clearing allowances to zero before setting a new non-zero value.

Technical Details

`UniswapV4Connector.addLiquidity()` directly sets approvals to the maximum value without first resetting them to zero.

This approach is problematic for non-standard tokens like USDT, which will revert if an existing non-zero allowance is directly modified to another non-zero value. Other contracts of the codebase, like `NftZapLib.sol`, correctly implement the safe approval pattern.

Impact

Medium. If a user attempts to add liquidity to a pool that contains non-standard tokens, such as USDT, the transaction will fail if there is an existing non-zero allowance. This prevents users from successfully adding liquidity to certain pools, blocking a core function of the protocol for popular stablecoin pairs.

Recommendation

Avoid approving if there is already an approval for that token. This also saves some gas.

Developer Response

Fixed using an alternative approach in commit 4d9f8c8.

2. Medium - `CurvePoolConnector.getPoolPrice()` can return price not in 18 decimals fixed precision

`CurvePoolConnector.getPoolPrice()` fetches the price between two assets in a Curve pool and returns such value in `18` decimals of precision.

Technical Details

The highlighted method requests a quote for the amount of output tokens given by a swap when 1e18 input tokens are provided. If such a request returns `0`, the method attempts to fetch the price by requesting a quote for the inverse swap. If this second quote is different from `0`, it inverts it to obtain the initial quote.

When inverting the second quote, the method does so by calculating:

```
85:             if (amountOut1 == 0) {
86:                 revert InvalidPrice();
87:             }
88:             price = 1e36 / amountOut1;
```

Because a Curve pool's `get_dy` method returns a raw amount of tokens, `amountOut1` is expressed in the fixed decimal precision the output token itself uses. This implies that:

1. If `amountOut1` is expressed with `18` decimals of precision, `price` will have precisely `18` decimals of precision, as expected.

2. If **`amountOut1`** is **not** expressed with **`18`** decimals of precision, **`price`** will be expressed in **`18 - baseToken.decimals()`**.
    - In the case in which **`baseTokens.decimals() > 18`**, **`price`** risks being rounded down to **`0`**, making the transaction revert with an **`InvalidPrice()`** error.

Impact

Medium. The highlighted method will return a value with incorrect decimal precision when requesting an inverse price in a non-standard decimal precision.

Recommendation

Refactor the inversion to use **`18 + baseToken.decimals()`** in the numerator so that **`price`** is always expressed in **`18`** decimals:

```
@@ -14,6 +14,10 @@ struct CurveSwapExtraData {
     int128 j;
 }

+interface IERC20Decimals {
+    function decimals() external view returns(uint256);
+}
+
 contract CurvePoolConnector is ILiquidityConnector {
     function addLiquidity(
         AddLiquidityParams memory addLiquidityParams
@@ -85,7 +89,10 @@ contract CurvePoolConnector is ILiquidityConnector {
             if (amountOut1 == 0) {
                 revert InvalidPrice();
             }
-            price = 1e36 / amountOut1;
+            address baseToken = ICurvePool(lpToken).coins(baseTokenIndex);
+            uint256 baseTokenDecimals = IERC20Decimals(baseToken).decimals();
+
+            price = 10 ** (18 + baseTokenDecimals) / amountOut1;
         }

         if (price == 0) {
```

Developer Response

Fixed in commits [296d744](296d744) and [2424a9d](2424a9d).

3. Medium - Unimplemented `UniswapV4Connector.swapExactTokensForTokens()` leads `SwapLib.swap()` to silently fail

`UniswapV4Connector.swapExactTokensForTokens()` implementation is empty. This will cause swaps in zap operations to fail silently.

Technical Details

When performing a zap operation `NftZapLib.zapIn()` delegates to `SwapLib.swap()`.

When `SwapLib.swap()` delegates to `UniswapV4Connector.swapExactTokensForTokens()`, the operation simply does nothing instead of reverting or performing the expected swap.

As a result, `NftZapLib.zapIn()` will proceed without reverting if `addLiquidityParams.amount0Desired` and `addLiquidityParams.amount1Desired` are set to zero.

Impact

Medium. Functionality is broken or can have unexpected behavior when using `Uniswap4Connector` as a swap router.

Recommendation

If you don't want to implement `swapExactTokensForTokens()` for UniswapV4 connector, revert when this function is called. Swaps in `zapIn()` should be performed through other connectors.

```
function swapExactTokensForTokens(
    SwapParams memory swap
) external payable virtual override {
    revert NotImplemented();
}
```

Developer Response

Fixed in commit 5205655.

4. Medium - `UniswapV3Connector.swapExactTokensForTokens()` breaks when using non-standard tokens like USDT.

`UniswapV3Connector.swapExactTokensForTokens()` implements the functionality for a Sickle to execute a swap on Uniswap V3 pools via its `SwapRouter` contract, using its `exactInput()` method.

Technical Details

Connectors that implement the `ILiquidityConnector` interface may be used by the `SwapLib` library to execute one or many swaps on the connector's target protocol.
Since `SwapLib._swap()` correctly handles and sets token approvals for the Sickle before calling a connector's swap function, connectors generally do not need to manage approvals themselves. This is the case for `UniswapV2Connector`, which does not contain any calls to `IERC20.approve()`.
However, this is not true for `UniswapV3Connector`: its `swapExactTokensForTokens()` method explicitly approves the swap router before performing the swap.

```
86:     function swapExactTokensForTokens(
87:         SwapParams memory swap
88:     ) external payable virtual override {
89:         UniswapV3SwapExtraData memory extraData =
90:             abi.decode(swap.extraData, (UniswapV3SwapExtraData));
91:
92:         IERC20(swap.tokenIn).approve(swap.router, swap.amountIn);
93:
94:         ISwapRouter(swap.router).exactInput(
95:             ISwapRouter.ExactInputParams({
96:                 path: extraData.path,
97:                 recipient: address(this),
98:                 deadline: block.timestamp,
99:                 amountIn: swap.amountIn,
100:                amountOutMinimum: swap.minAmountOut
101:            })
102:        );
103:    }
```

As a result, given that a non-zero allowance was already set by `SwapLib`, when using tokens that require allowances to be reset to `0` before being written to, like USDT, the transaction will fail.

Additionally, given that `IERC20.approve()` is used expecting a `bool` return value, using non-standard tokens that do not return data for such ca all will also cause the transaction

to fail.

As a result, because `SwapLib` has already set a non-zero allowance, transactions involving tokens that require allowances first to be reset to `0` before being updated — such as USDT — will fail.
Furthermore, since `IERC20.approve()` is called with the expectation that it returns a `bool`, using non-standard tokens that do not return any data from `approve()` will also cause the transaction to fail.

Impact

Medium. `UniswapV3Connector.swapExactTokensForTokens` will malfunction when using non-standard tokens.

Recommendation

Conform to the standard connector behavior of not setting an allowance, given that it is already handled by `SwapLib`.

Developer Response

Fixed in commit 39a80a4.

Low Findings

1. Low - Different representations of native `ETH` could lead to transaction failure

The `TransferLib` contract handles two different representations of native `ETH`:

1. `ETH` (0xEeeeeEeeeEeEeeEeEeEeeEEEeeeeEeeeeeeeEEeE)
2. `UNISWAP_ETH` (0x0000000000000000000000000000000000000000)

While the contract accepts both representations, there is no check preventing their simultaneous use in the same transaction. In this scenario, the `_checkMsgValue()` check in `transferTokensFromUser()` will fail.

Impact

Low. While this may not be a usual scenario, using both tokens simultaneously can lead to transaction failure.

Recommendation

Add a check in the `checkTransfersFrom()` modifier to prevent using both `ETH` and `UNISWAP_ETH` in the same transaction:

```
modifier checkTransfersFrom(
    address[] memory tokensIn,
    uint256[] memory amountsIn
) {
    uint256 tokenLength = tokensIn.length;
    if (tokenLength != amountsIn.length) {
        revert ArrayLengthMismatch();
    }
    if (tokenLength == 0) {
        revert TokenInRequired();
    }
    for (uint256 i; i < tokenLength; i++) {
        if (amountsIn[i] == 0) {
            revert AmountInRequired();
        }
    }
    bool hasETH = false;
    bool hasUniswapETH = false;
    for (uint256 i; i < tokenLength; i++) {
        if (tokensIn[i] == ETH) {
            hasETH = true;
        }
        if (tokensIn[i] == UNISWAP_ETH) {
            hasUniswapETH = true;
        }
        if (hasETH && hasUniswapETH) {
            revert SameTokenIn();
        }
        for (uint256 j = i + 1; j < tokenLength; j++) {
            if (tokensIn[i] == tokensIn[j]) {
                revert SameTokenIn();
            }
        }
    }
    _;
}
```

Developer Response

Fixed in commit 0076a77.

2. Low - Missing configuration validations in settings registries

Some missing validations could lead to configurations that pass checks but fail during execution when automated operations are triggered.

Technical Details

The `checkPositionSettings()` modifier in `PositionSettingsRegistry` and `checkConfigValues()` modifier in `NftSettingsRegistry` lack some validations.

These modifiers do not properly verify that exit tokens are valid tokens from the pool, which could lead to failed transactions or unexpected behavior during automated exits.

Additionally, in the `NftSettingsRegistry`, there's a missing order check for trigger ticks.

Impact

Low. Several validations are missing in the configuration checks, which could lead to failures in automated operations.

Recommendation

Add comprehensive token validation in `checkPositionSettings()`:

```
    // ...
    if (settings.autoExit) {
        // ...

        ILiquidityConnector connector = ILiquidityConnector(
            _connectorRegistry.connectorOf(address(settings.router))
        );
        uint256[] memory reserves =
            connector.getReserves(address(settings.pool));
        if (
            settings.exitConfig.baseTokenIndex >= reserves.length
                || settings.exitConfig.quoteTokenIndex >= reserves.length
                || settings.exitConfig.baseTokenIndex
                    == settings.exitConfig.quoteTokenIndex
        ) {
            revert InvalidTokenIndices();
        }
```

```
        if (settings.exitConfig.triggerReservesLow.length > 0) {
            if (
                reserves.length
                    != settings.exitConfig.triggerReservesLow.length
            ) {
                revert InvalidTriggerReserves();
            }
        }
+
+       address[] memory tokens = connector.getTokens(address(settings.pool));
+
+       bool isLowTokenValid = false;
+       bool isHighTokenValid = false;
+
+       for (uint256 i = 0; i < tokens.length; i++) {
+           if (tokens[i] == settings.exitConfig.exitTokenOutLow) {
+               isLowTokenValid = true;
+           }
+           if (tokens[i] == settings.exitConfig.exitTokenOutHigh) {
+               isHighTokenValid = true;
+           }
+       }
+
+       if (!isLowTokenValid) revert InvalidExitTokenLow();
+       if (!isHighTokenValid) revert InvalidExitTokenHigh();
    } else {
    // ...
```

Add comprehensive token validation in **`checkConfigValues()`**:

```
    // ...
    if (!settings.autoExit) {
        if (
            settings.exitConfig.triggerTickLow != 0
                || settings.exitConfig.triggerTickHigh != 0
                || settings.exitConfig.exitTokenOutLow != address(0)
                || settings.exitConfig.exitTokenOutHigh != address(0)
                || settings.exitConfig.slippageBP != 0
                || settings.exitConfig.priceImpactBP != 0
        ) {
            revert AutoExitNotSet();
        }
```

```
    } else {
        if (
            settings.exitConfig.triggerTickLow == 0
                && settings.exitConfig.triggerTickHigh == 0
        ) {
            revert ExitTriggersNotSet();
        }
+       if (settings.exitConfig.triggerTickLow >=
settings.exitConfig.triggerTickHigh) {
+           revert InvalidExitTriggers();
+       }
        if (settings.exitConfig.slippageBP > MAX_SLIPPAGE_BP) {
            revert InvalidSlippageBP();
        }
        if (
            settings.exitConfig.priceImpactBP > MAX_PRICE_IMPACT_BP
                || settings.exitConfig.priceImpactBP == 0
        ) {
            revert InvalidPriceImpactBP();
        }
+
+       INftLiquidityConnector connector = INftLiquidityConnector(
+           connectorRegistry.connectorOf(address(key.nftManager))
+       );
+       NftPoolInfo memory poolInfo = connector.poolInfo(address(settings.pool),
settings.poolId);
+       if (settings.exitConfig.exitTokenOutLow != poolInfo.token0 &&
+           settings.exitConfig.exitTokenOutLow != poolInfo.token1) {
+           revert InvalidExitTokenLow();
+       }
+       if (settings.exitConfig.exitTokenOutHigh != poolInfo.token0 &&
+           settings.exitConfig.exitTokenOutHigh != poolInfo.token1) {
+           revert InvalidExitTokenHigh();
+       }
+   }
    _;
}
```

Developer Response

Fixed. Added >= check here [38ae09d](). Exit tokens do not need to be in the pool as we swap for them.

3. Low - `TransferLib.transferTokenFromUser()` reverts when `UNISWAP_ETH` is used

`TransferLib.transferTokenFromUser()` is used to pull funds from a Sickle's owner into the Sickle itself.

Technical Details

`TransferLib.transferTokenFromUser()` supports both native ETH and ERC20 tokens: in the case of ETH, the method expects `msg.value` to match the `amountIn` parameter by using the internal `_checkMsgValue()` method.

The highlighted function fails to consider `UNISWAP_ETH` as the native currency and instead treats it as an ERC20 token. Because of this, the `_checkMsgValue()` enforces that `msg.value == 0`, reverting in case such a condition doesn't hold.

As a result, any invocation of `TransferLib.transferTokenFromUser()` with `tokenIn == UNISWAP_ETH && msg.value > 0` will revert.

PoC

Add the following to `test/libraries/TransferLib.t.sol` and execute with `forge t --mt test_transfer_token_from_user_has_unieth`

```
function test_transfer_token_from_user_has_unieth() public {
    address tokensIn = address(0); // == UNISWAP_ETH
    uint amountsIn = 10;

    deal(sickleAdmin, 100);

    vm.prank(sickleAdmin);
    vm.expectRevert(bytes4(keccak256("IncorrectMsgValue()")));
    ctx.transferLib.transferTokenFromUser{ value: 10 }(
        tokensIn, amountsIn, address(farmStrategy), Deposit
    );
}
```

Impact

Low. Although not currently used with the strategies intended for Uniswap V4, `TransferLib.transferTokenFromUser()` does not function properly when interacting with

Uniswap V4's native ETH representation.

Recommendation

Adapt the `_checkMsgValue()` call to also consider `UNISWAP_ETH` as native currency:

```
@@ -78,7 +78,7 @@ contract TransferLib is MsgValueModule, DelegateModule,
ITransferLib {
         address strategy,
         bytes4 feeSelector
     ) public payable checkTransferFrom(tokenIn, amountIn) {
-        _checkMsgValue(amountIn, tokenIn == ETH);
+        _checkMsgValue(amountIn, tokenIn == ETH || tokenIn == UNISWAP_ETH);

         _transferTokenFromUser(tokenIn, amountIn, strategy, feeSelector);
     }
```

Developer Response

Fixed in main.

4. Low - `rebalance()` can apply settings to the wrong token due to incorrect `tokenId` handling

Technical Details

In the current implementation, `NftFarmStrategy.rebalance()` retrieves the token ID using `getTokenId()` in both `UniswapV4Connector.sol` and `UniswapV3Connector.sol`.

The issue is that:

- `UniswapV4Connector.getTokenId()` just returns the most recently created token by anyone:

```
function getTokenId(
    address,
    address // owner completely ignored
) external view virtual override returns (uint256) {
    return positionManager.nextTokenId() - 1;
}
```

- `UniswapV3Connector.getTokenId()` returns the most recent token owned by the specified owner:

```
function getTokenId(
    address nft,
    address owner
) external view virtual returns (uint256) {
    return IERC721Enumerable(nft).tokenOfOwnerByIndex(
        address(owner), IERC721Enumerable(nft).balanceOf(address(owner)) - 1
    );
}
```

This holds only when `rebalance()` creates a new token in `_zap_in()` and this happens only when `params.increase.zap.tokenId` is zero.

If a sickle's owner calls `rebalance()` with that param different from zero, subsequent operations `_reset_nft_settings()` and `_deposit_nft()` are performed with:

- the newest created `tokenId` that can belong to any other user in the case of `UniswapV4Connector.sol`
- the newest created `tokenId` of the user in the case of `UniswapV3Connector.sol`

PoC

In `UniswapV4FarmTest.sol`:

```
function _test_rebalance(
    uint256 amountInGross
) internal {
    uint256 amountIn = amountInGross * 9991 / 10_000;

    _test_deposit_using_eth(amountInGross); // 12067 on Arbitrum
    uint256 tokenId1 = _get_token_id();

    _test_deposit_using_eth(amountInGross); // 12068 on Arbitrum
    uint256 tokenId2 = _get_token_id();

    _test_deposit_using_eth(amountInGross); // 12069 on Arbitrum
    uint256 tokenId3 = _get_token_id();
```

```
    vm.startPrank(sickleOwner);

    uint256 amountInGross2 = amountIn / 2; // deposit half this time
    uint256 amountIn2 = amountInGross2 * 9991 / 10_000;

    address[] memory tokensOut = new address[](1);
    tokensOut[0] = address(0);

    nftFarmStrategy.rebalance(
        NftRebalance(
            IUniswapV3Pool(address(contracts.positionManager)),
            // use the positionManager to get the rebalance fee
            NftPosition({
                farm: farm,
                nft: INonfungiblePositionManager(
                    address(contracts.positionManager)
                ),
                tokenId: tokenId1
            }),
            NftHarvest({
                harvest: defaultSimpleHarvest,
                swaps: new SwapParams[](0),
                outputTokens: new address[](0),
                sweepTokens: defaultSimpleHarvest.rewardTokens
            }),
            NftWithdraw({
                zap: _get_withdraw_zap_out_data(tokenId1, false),
                extraData: "",
                tokensOut: tokensOut
            }),
            NftIncrease({
                tokensIn: new address[](0),
                amountsIn: new uint256[](0),
                zap: _get_deposit_zap_in_data(amountIn2, tokenId2),
                extraData: ""
            })
        ),
        SWEEP_TOKENS
    );

    assertGt(
        ctx.registry.collector().balance,
        amountInGross * 9 / 10_000, // plus rebalance fee
```

```
        "ETH balance of collector"
    );

    // get last token id
    uint256 tokenIdRebalance = _get_token_id();

    // no new token was created during rebalance
    // tokenId2 was the target of the rebalance
    // tokenId3 was the target of setting reset
    assertEq(tokenIdRebalance, tokenId3);
}
```

Run with:

```
forge test --mp test/strategies/UniswapV4Arbitrum.t.sol --mt test_rebalance -vvvv
```

The console logs show the settings reset from the first token to the last one:

```
    |   |   |   |   |   ├─ emit NftSettingsUnset(key: NftKey({ sickle:
0xd5Ba94B49C4c80833cC5dE77C43BEab843961bFA, nftManager:
0xd88F38F930b7952f2DB2432Cb002E7abbF3dD869, tokenId: 12067 [1.206e4] }))
    |   |   |   |   |   ├─ emit NftSettingsSet(key: NftKey({ sickle:
0xd5Ba94B49C4c80833cC5dE77C43BEab843961bFA, nftManager:
0xd88F38F930b7952f2DB2432Cb002E7abbF3dD869, tokenId: 12069 [1.206e4] }),
settings: NftSettings({ pool: 0x360E68faCcca8cA495c1B759Fd9EEe466db9FB32, poolId:
0x864abca0a6202dba5b8868772308da953ff125b0f95015adbf89aaf579e903a8,
autoRebalance: false, rebalanceConfig: RebalanceConfig({ tickSpacesBelow: 1,
tickSpacesAbove: 1, bufferTicksBelow: 0, bufferTicksAbove: 0, dustBP: 100,
priceImpactBP: 200, slippageBP: 0, cutoffTickLow: 0, cutoffTickHigh: 0, delayMin:
0, rewardConfig: RewardConfig({ rewardBehavior: 2, harvestTokenOut:
0x0000000000000000000000000000000000000000 }) }), automateRewards: false,
rewardConfig: RewardConfig({ rewardBehavior: 2, harvestTokenOut:
0x0000000000000000000000000000000000000000 }), autoExit: false, exitConfig:
ExitConfig({ triggerTickLow: 0, triggerTickHigh: 0, exitTokenOutLow:
0x0000000000000000000000000000000000000000, exitTokenOutHigh:
0x0000000000000000000000000000000000000000, priceImpactBP: 0, slippageBP: 0 }),
extraData: 0x }))
```

Impact

Low. When a user calls `rebalance()`, their NFT settings are removed from their token and may be applied to another token that is not involved in the rebalance or to a non-existent key.

Recommendation

Ensure that `params.increase.zap.tokenId` is always zero when calling `rebalance()`.

Developer Response

Partially fixed in commit d03df71.

5. Low - Misleading event emissions in `NftFarmStrategy` when used with Uniswap V3/V4 connectors

Technical Details

The `NftFarmStrategy` contract emits events after calling connector functions, regardless of whether those connector functions perform any operations.

This happens in `_deposit_nft()` and `_withdraw_nft()` when the connector is the `UniswapV4Connector` or the `UniswapV3Connector`, where both `depositExistingNft()` and `withdrawNft()` are empty.

This mismatch creates a situation where events are emitted, suggesting that staking or unstaking operations took place when, in reality, no such operations occurred.

Additionally, gas is wasted for the external delegate calls to the connectors.

Impact

Low. This issue can lead to misleading events, potentially generating confusion for users interacting with the system through a front-end that relies on these events or errors in monitoring and analytics systems that track user positions.

Recommendation

Conditionally call `_deposit_nft()` and `_withdraw_nft()` based on the connector type.

Add a revert for those not implemented functions in `UniswapV4Connector.sol`:

```
function depositExistingNft(
    NftPosition calldata, // position,
```

```
    bytes calldata // extraData
) external payable virtual override {
    revert NotImplemented();
}

function withdrawNft(
    NftPosition calldata, // position,
    bytes calldata // extraData
) external payable virtual override {
    revert NotImplemented();
}
```

Developer Response

Acknowledged. The present solution was decided as the better one (no-op vs conditional). It seems better for strategies to remain connector agnostic and to keep branching low. The extra events are also not particularly misleading because the NFT was "deposited" to the Sickle, not for any further staking contract.

6. Low - `UniswapV4Connector.addLiquidity()` uses different values as slippage tolerance when minting and increasing a position

`UniswapV4Connector.addLiquidity()` increases a Sickle's Uniswap V4 liquidity position, either by minting a new position via `_mint()` or by improving an existing position's liquidity via `_increase_liquidity()`.

Technical Details

While `_mint()` uses `extraData.amount0Max` and `extraData.amount1Max` as the maximum amount of tokens that may be provided to the Uniswap pool, `_increase_liquidity` uses `addLiquidityParams.amount0Desired` and `addLiquidityParams.amount1Desired`.

Impact

Low.

Recommendation

Use `extraData.amount0Max` and `extraData.amount1Max` also within `_increase_liquidity`.

Developer Response

Fixed in commit fa6837a, using `amount0Desired` instead.

7. Low - Inconsistent upgrade pattern between `NftSettingsRegistry` and `PositionSettingsRegistry`

The `PositionSettingsRegistry` and `NftSettingsRegistry` contracts have inconsistent upgrade patterns for handling the `ConnectorRegistry`. While these contracts serve similar purposes and should maintain consistent behavior, they implement fundamentally different approaches to registry management.

Technical Details

`PositionSettingsRegistry` uses a mutable approach with timelock protection:

```
ConnectorRegistry private _connectorRegistry;

function setConnectorRegistry(
    ConnectorRegistry connectorRegistry
) external onlyTimelockAdmin {
    _connectorRegistry = connectorRegistry;
    emit ConnectionRegistrySet(address(connectorRegistry));
}
```

`NftSettingsRegistry` uses an immutable approach:

```
ConnectorRegistry public immutable connectorRegistry;

constructor(SickleFactory _factory, ConnectorRegistry _connectorRegistry) {
    factory = _factory;
    connectorRegistry = _connectorRegistry;
}
```

In the event of a needed upgrade in the `ConnectorRegistry`, only the `PositionSettingsRegistry` could be updated, while the `NftSettingsRegistry` would remain linked to the outdated or vulnerable registry. A complete redeployment would be required, which could disrupt user settings.

Impact

Low.

Recommendation

Adopt a consistent approach to registry management across both contracts.

Developer Response

Fixed in commit 684f648.

8. Low - `UniswapV3Connector` may fail to burn NFT positions due to residual owed tokens

`UniswapV3Connector.removeLiquidity()` may fail to burn NFT positions due to residual owed tokens even if all liquidity is removed.

Technical Details

`UniswapV3Connector.removeLiquidity()` is designed to handle the complete process of removing liquidity in a single transaction, including the possible burning of the NFT token when all liquidity is removed.

The issue is that even if `currentLiquidity == 0`, the call to `burn` might still fail if there are tokens left in the position's `tokensOwed0` or `tokensOwed1` fields. This occurs because the Uniswap V3 `NonfungiblePositionManager.burn()` verifies that also `tokensOwed0` and `tokensOwed1` are zero before allowing the burn operation.

If the `amount0Max` or `amount1Max` parameters passed to `_collect()` are not large enough to collect all accumulated tokens, these residual tokens will prevent the NFT from being burned and tx fails.

Impact

Low. This scenario can lead to failed transactions.

Recommendation

Modify `removeLiquidity()` to collect all fees:

```
    function removeLiquidity(
        NftRemoveLiquidity memory removeLiquidityParams
    ) external override {

        // ...

        _collect(
            removeLiquidityParams.nft,
```

```
            removeLiquidityParams.tokenId,
-           removeLiquidityParams.amount0Max,
-           removeLiquidityParams.amount1Max
+           type(uint128).max,
+           type(uint128).max
        );

        position = positionInfo(
            address(removeLiquidityParams.nft), removeLiquidityParams.tokenId
        );
        currentLiquidity = position.liquidity;
        if (currentLiquidity == 0) {
            removeLiquidityParams.nft.burn(removeLiquidityParams.tokenId);
        }
    }
```

Or check if there are owed tokens left before the burn operation:

```
    function removeLiquidity(
        NftRemoveLiquidity memory removeLiquidityParams
    ) external override {

        // ...

        position = positionInfo(
            address(removeLiquidityParams.nft), removeLiquidityParams.tokenId
        );
        currentLiquidity = position.liquidity;
-       if (currentLiquidity == 0) {
+       if (currentLiquidity == 0 && position.tokensOwed0 == 0 &&
position.tokensOwed1 == 0) {
            removeLiquidityParams.nft.burn(removeLiquidityParams.tokenId);
        }
    }
```

Developer Response

Acknowledged. A failure here is acceptable since no transaction would take place. We use max_uint128 at the calling site at the moment.

Gas Saving Findings

1. Gas - Optimize nested loops in `TransferLib.checkTransfersFrom()`

`TransferLib.checkTransfersFrom()` verifies that `address[] tokensIn` doesn't contain duplicate entries with two nested loops, which runs with quadratic complexity in the array's length.

Technical Details

The current duplicate check can be optimized to run in linear complexity and require a single pass of the entire array by verifying the array's elements are strictly increasing.

Impact

Gas savings.

Recommendation

Refactor the function as follows:

```
@@ -167,12 +167,8 @@ contract TransferLib is MsgValueModule, DelegateModule,
ITransferLib {
                 revert AmountInRequired();
             }
         }
-        for (uint256 i; i < tokenLength; i++) {
-            for (uint256 j = i + 1; j < tokenLength; j++) {
-                if (tokensIn[i] == tokensIn[j]) {
-                    revert SameTokenIn();
-                }
-            }
+        for (uint256 i; i < tokenLength - 1; i++) {
+            if (tokensIn[i] >= tokensIn[i + 1]) revert SameTokenIn();
         }
         _;
     }
```

Note that, in order for this recommendation to function appropriately, the `tokensIn` array must be sorted in increasing order when provided as a transaction's calldata.

Developer Response

Acknowledged. The gas cost is minimal due to the size of the array (2-3 items) so it seems better than having an offchain requirement to send them in order.

Informational Findings

1. Informational - Misleading function name in `NftSettingsRegistry.sol`

Technical Details

`resetNftSettings()` in the `NftSettingsRegistry.sol` contract has a name that suggests it resets NFT settings to default or empty values. However, the function does not reset settings to default values. Instead, it is used to move the settings from the old key to the new key after a rebalance.

This is a transfer of settings from one key to another, not a reset to default values.

The misleading name could lead to confusion for developers and auditors, who expect the function to reset settings to default values rather than migrate them to a new key.

Impact

Informatioanl.

Recommendation

Rename the function to better reflect its actual behavior, like `transferNftSettings()`.

Developer Response

Fixed in commit c3596f7.

2. Informational - `TransferLib.chargeFees()` ignores internal calls's return values

`TransferLib.chargeFees()` batches calls to the `TransferLib.chargeFee()` public method.

Technical Details

While `TransferLib.chargeFee()` returns a number of tokens to which the protocol's fees have been subtracted, the batch method `TransferLib.chargeFees()` doesn't return any data.

Impact

Informational.

Recommendation

Modify `TransferLib.chargeFees()` to return a `uint256[] remainders` array, populated with the return data of each internal call to `TransferLib.chargeFee()`:

Developer Response

Acknowledged. This is not currently needed.

3. Informational - Use a constant instead of a hardcoded address for native

Technical Details

In `addLiquidity()` in the `UniswapV4Connector.sol` contract, the zero address `address(0)` is used directly in the code to check if a token is `ETH`, instead of using a dedicated constant. This approach reduces code readability and can lead to confusion, especially since other parts of the codebase, such as `TransferLib.sol`, have specific constants defined for this purpose.

Recommendation

Use a dedicated constant for native handling.

Impact

Informational.

Developer Response

Fixed in commit 6e9e5b0.

4. Informational - `CurvePoolConnector.addLiquidity()` and `CurvePoolConnector.removeLiquidity()` can be simplified

`CurvePoolConnector.addLiquidity()` and `CurvePoolConnector.removeLiquidity()` are used to make a Sickle provide and remove liquidity from Curve pools.

Technical Details

Both methods read elements from a calldata `uint256[]` array and needlessly copy them to a memory array to use in an external call to Curve's pool.

- `CurvePoolConnector.sol#L22-L26` copies elements from `addLiquidityParams.desiredAmounts` into `amounts`.

- `CurvePoolConnector.sol#L42-L47` copies elements from `removeLiquidityParams.minAmountsOut` into `minAmounts`.

Impact

Informational.

Recommendation

Avoid copying elements manually and use the calldata arrays directly.

Developer Response

Fixed in commits 5e83d39 and 6666bed.

5. Informational - Avoid redundant token approvals in `zapIn()`

Technical Details

`NftZapLib.zapIn()` approves tokens for the router that are not necessary for the UniswapV4Connector.

`UniswapV4Connector.addLiquidity()` performs its own token approvals.

This results in redundant approval operations that waste gas and add unnecessary complexity to the codebase.

Impact

Informational.

Recommendation

Centralize all token approvals in `NftZapLib` and remove them from the connector implementations. Add a connector type identifier to the `NftZapIn` struct to identify the connector.

```
enum ConnectorType {
    Default,
    UniswapV4,
    // Other connector types
}

struct NftZapIn {
    SwapParams[] swaps;
```

```
    NftAddLiquidity addLiquidityParams;
    ConnectorType connectorType;
}
```

Then, modify `NftZapLib.zapIn()` to handle specialized approvals based on the connector type.

Developer Response

Acknowledged. `UniswapV4Connector` is the only connector that has a custom approval system, so we decided to maintain it this way (make v4 cumbersome instead of affecting the rest of the system, creating repetitive approvals in a dozen contracts).

6. Informational - Unnecessary operation sequence in `UniswapV4Connector.removeLiquidity()`

In `UniswapV4Connector`, `removeLiquidity()` performs unnecessary operations by calling both `_collect()` and `_decrease_liquidity()` sequentially.

Technical Details

`removeLiquidity()` first calls `_collect()` to gather fees, followed by `_decrease_liquidity()` to remove liquidity.

```
// It seems that we need to collect first
_collect(
    IPositionManager(address(removeLiquidityParams.nft)),
    removeLiquidityParams.tokenId,
    removeLiquidityParams.extraData
);

_decrease_liquidity(removeLiquidityParams);
```

This sequence is unnecessary and inefficient because in Uniswap V4's architecture a single decrease operation with liquidity > 0 automatically:

- Calculates all accumulated fees on the position
- Removes the specified amount of liquidity
- Returns both the removed liquidity and accumulated fees

The `_collect()` operation is effectively a `DECREASE_LIQUIDITY` action with liquidity = 0. When `_decrease_liquidity()` is called afterward, it performs another operation that would already include any remaining fees.

This is also evident in the underlying Uniswap V4 `PositionManager._decrease()` calls `PositionManager._modifyLiquidity()` which returns both the `liquidityDelta` and `feesAccrued`.

Fees are returned to `PositionManager` by `PoolManager.modifyLiquidity()` and calculated in `Pool.modifyLiquidity()`.

The redundant operations increase gas costs by making two separate contract calls and unnecessary complexity to the implementation.

Impact

Informational.

Recommendation

Simplify the implementation by removing the redundant `_collect()` call.

Developer Response

Fixed in commit 592a973. Now using burn or decrease accordingly.

7. Informational - Unused functions

Remove unused functions to reduce contract size and deployment costs.

Technical Details

- `UniswapV3Connector._claim_fees()` isn't invoked anywhere in the contract.

Impact

Informational.

Recommendation

Remove the highlighted methods.

Developer Response

Fixed in commit eda0482.

8. Informational - Unnecessary addition is executed when setting `deadline` for Uniswap V4 interactions

`UniswapV4Connector` interacts with Uniswap V4 by invoking its `PositionManager.modifyLiquidities()` method, which expects a `deadline` parameter to be provided.

Technical Details

Just like in V3, Uniswap V4 allows for `deadline = block.timestamp` to be provided, as the implementation of the `checkDeadline()` modifier suggests.

Impact

Informational.

Recommendation

Provide `deadline = block.timestamp` instead of `deadline = block.timestamp + 1` when calling `PositionManager.modifyLiquidities`.

Developer Response

Fixed in commit 05baa1d.

9. Informational - Implement hooks validation in `UniswapV4Connector`

The `UniswapV4Connector` contract accepts and processes arbitrary hooks without any validation mechanism.

Technical Details

In the current implementation, hooks are passed directly through `extraData` and used without validation regardless of its security:

- `_mint()`
- `_increase_liquidity()`
- `_decrease_liquidity()`
- `collect()`

Apart from security concerns, there can also be some hooks that are incompatible with the current mechanisms, potentially leading to unexpected behavior or failures.

Impact

Informational.

Recommendation

Implement a **`HookRegistry`** contract to manage and validate hooks:

```
interface IHookRegistry {
    error InvalidAddress();
    error HookNotEnabled();

    event HookEnabled(address indexed hook);
    event HookDisabled(address indexed hook);

    function enableHook(address hook) external;
    function disableHook(address hook) external;
}

contract HookRegistry is IHookRegistry {
    mapping(address => bool) public enabledHooks;

    modifier onlyEnabledHooks(address hook) {
        if (hook != address(0) && !enabledHooks[hook]) revert HookNotEnabled();
        _;
    }

    function enableHook(address hook) external {
        if (hook == address(0)) revert InvalidAddress();
        enabledHooks[hook] = true;
        emit HookEnabled(hook);
    }

    function disableHook(address hook) external {
        enabledHooks[hook] = false;
        emit HookDisabled(hook);
    }
}
```

Then modify the **`UniswapV4Connector`** to use this registry:

```
contract UniswapV4Connector is INftLiquidityConnector, INftFarmConnector,
HookRegistry {
```

```
    // ...
```

Developer Response

Acknowledged. This is an as-yet-unknown user error risk rather than a security concern, will be revisited once hooks are in widespread use and the risk can be quantified.

10. Informational - `UniswapV4Connector.removeLiquidity()` uses hardcoded `hookData`

`UniswapV4Connector.removeLiquidity()` implements functionality for a Sickle to remove liquidity from Uniswap V4 pools.

Technical Details

After removing liquidity, the highlighted method will attempt to burn the position's NFT if its entire liquidity has been withdrawn:

```
135:         currentLiquidity = _get_current_liquidity(removeLiquidityParams);
136:
137:         if (currentLiquidity == 0) {
138:             bytes memory actions =
139:                 abi.encodePacked(uint8(Actions.BURN_POSITION));
140:             bytes[] memory params = new bytes[](1);
141:             params[0] = abi.encode(
142:                 removeLiquidityParams.tokenId,
143:                 uint128(0),
144:                 uint128(0),
145:                 new bytes(0) // AUDIT hookData = ""
146:             );
147:             IPositionManager(address(removeLiquidityParams.nft))
148:                 .modifyLiquidities(abi.encode(actions, params),
block.timestamp + 1);
149:         }
```

The call to `modifyLiquidities()` is always executed with an empty bytes array for `hookData`.

Impact

Informational. Depending on the pool that's being interacted with, the `hookData` field may be helpful to Sickle: hardcoding its value might limit future integrations, requiring a

redeployment of the connector.

Recommendation

Add an extra `bytes burnHookData` field to the `NftRemoveLiquidity` struct and use such value instead of `new bytes(0)` when burning a Uniswap V4 position NFT.

Developer Response

Fixed in commit b97b7b6.

11. Informational - Silent overflow possible in calls to `PositionManager.modifyLiquidities()`

Different calls to Uniswap V4's `PositionManager.modifyLiquidities()` provide parameters with larger types than expected, which may lead to silent overflows during such interactions.

Technical Details

Within `UniswapV4Connector`, the following methods provide `uint256` parameters to calls that expect `uint128`s to be provided:

1. `_increase_liquidity()`: triggers the `INCREASE_LIQUIDITY` action, which expects the third and fourth parameters (`amount0Max` and `amount1Max`) to be 128 bit unsigned integers. The method provides `addLiquidityParams.amount0Desired` and `addLiquidityParams.amount1Desired`, which are both 256-bit unsigned integers.
2. `_mint()`: triggers the `MINT_POSITION` action, which expects the fifth and sixth parameters (`amount0Max` and `amount1Max`) to be 128 bit unsigned integers. The method provides `extraData.amount0Max` and `extraData.amount0Max`, both of which are 256-bit unsigned integers.
3. `_decrease_liquidity()`: triggers the `DECREASE_LIQUIDITY` action, which expects the third and fourth parameters (`amount0Min` and `amount1Min`) to be 128 bit unsigned integers. The method provides `removeLiquidityParams.amount0Min` and `removeLiquidityParams.amount1Min`, both of which are 256-bit unsigned integers.

Specifications on the expected types can be found in the Uniswap documentation:

- Mint position
- Increase liquidity

- Decrease liquidity

Impact

Informational.

Recommendation

Ensure that the highlighted fields fit within a `uint128` and cast them down.

Developer Response

Fixed in commit 29d2b2e.

Final Remarks

Sickle's modular architecture is built around adaptability, offering flexibility when it comes to enabling or disabling third-party integrations. The registry, library, connector, and lens abstractions effectively represent and separate the various components required for a smart wallet account to interact with virtually any protocol.

Key breaking changes introduced in Uniswap V4, such as the reintegration of native ETH within the protocol or the action-based interaction pattern used in their `PositionManager` contract, led to issues in Sickle's `UniswapV4Connector`. Furthermore, the potential impact of Uniswap V4 hooks on the protocol's functionality was not extensively addressed, as the Sickle team intends to evaluate such integrations individually on a case-by-case basis.

A lack of strict testing for Sickle's `CurvePoolConnector.getPoolPrice()` method led it to return incorrect values when interacting with Curve pools that contain ERC20 tokens with non-standard fixed-point precision.

Overall, Sickle is built on solid foundations, and its team has been responsive throughout the entire review. Their commitment to addressing all raised issues and applying improvements to the in-scope assets ensures the platform's future security.