V4 Swap Router Security Review

Auditors
Kaden, Security Researcher

March 11, 2025

1 Executive Summary

Over the course of 5 days in total, Kaden was engaged to review V4 Swap Router.

Metadata

Repository	Commit
v4-router	d047e0c

Summary

Type of Project	AMM
Timeline	Feb 12, 2025 - Feb 16, 2025
Methods	Manual Review

Total Issues

Critical Risk	1
High Risk	0
Medium Risk	2
Low Risk	3
Informational	2
Gas Optimizations	1

Contents

1 Executive Summary **1**

2 Introduction **3**

3 Findings **3**

3.1 Critical Risk . . . 3

3.1.1 Approvals can be stolen by providing an arbitrary `payer` in low-level `swap` function . . . 3

3.2 Medium Risk . . . 5

3.2.1 `outputAmount` may be incorrectly computed during multihop exact input swaps . . . 5

3.2.2 Multihop swaps return an incorrect `BalanceDelta` . . . 7

3.3 Low Risk . . . 8

3.3.1 Exclusive `amountLimit` used . . . 8

3.3.2 Exact input swaps may not use the full input amount, potentially leaving ETH in the contract . . . 9

3.3.3 Fee-on-transfer tokens are not supported . . . 11

3.4 Informational . . . 11

3.4.1 `IUniswapV4Router04.sol` compiler warning . . . 11

3.4.2 ETH is not synced prior to settlement . . . 12

3.5 Gas Optimizations . . . 13

3.5.1 Efficient encoding order . . . 13

2 Introduction

V4 Swap Router is a simple and optimized router for swapping on Uniswap V4 with an ABI inspired by UniswapV2Router02.

The focus of the security review was on the smart contracts in the `src/` directory.

Disclaimer: This review does not make any warranties or guarantees regarding the discovery of all vulnerabilities or issues within the audited smart contracts. The auditor shall not be liable for any damages, claims, or losses incurred from the use of the audited smart contracts.

3 Findings

3.1 Critical Risk

3.1.1 Approvals can be stolen by providing an arbitrary `payer` in low-level `swap` function

Severity: Critical

Context: UniswapV4Router04.sol#L217-L226

Description:

During settlement of account deltas after a swap, we transfer funds from the `data.payer` account:

```
inputCurrency.settle(poolManager, data.payer, inputAmount, input6909);
```

```
function settle(Currency currency, IPoolManager manager, address payer, uint256 amount,
↪  bool burn) internal {
    // for native currencies or burns, calling sync is not required
    // short circuit for ERC-6909 burns to support ERC-6909-wrapped native tokens
    if (burn) {
        manager.burn(payer, currency.toId(), amount);
    } else if (currency.isAddressZero()) {
        manager.settle{value: amount}();
    } else {
        manager.sync(currency);
        if (payer != address(this)) {
            IERC20Minimal(Currency.unwrap(currency)).transferFrom(payer,
            ↪  address(manager), amount);
        } else {
            IERC20Minimal(Currency.unwrap(currency)).transfer(address(manager),
            ↪  amount);
        }
        manager.settle();
    }
}
```

For each externally accessible swap method, we provide the `msg.sender` as the `data.payer`, except for the low-level `swap` method, where any arbitrary `data` can be provided:

```
function swap(bytes calldata data, uint256 deadline)
    public
    payable
    virtual
    override(IUniswapV4Router04)
    checkDeadline(deadline)
    returns (BalanceDelta)
{
    return _unlockAndDecode(data);
}
```

This allows an attacker to execute a low-level `swap`, providing any address that has a token allowance, either ERC20 or ERC6909, to the router as the `data.payer` to steal tokens directly from their wallet to execute an arbitrary swap.

Note that this attack also affects `permit2` signatures which can be frontran to steal permitted tokens in the same manner.

Recommendation:

All external swap functions must set the `data.payer` as the `msg.sender`, or use the `msg.sender` directly for settlement logic. If it's desirable to add functionality to allow users to execute swaps on behalf of others, approval/operator logic must be implemented directly on the router contract.

v4-router:

To resolve this issue, we now apply an efficient check in the low-level swap function by extracting the expected position of the `data.payer` in calldata with assembly code, and then asserting it matches `msg.sender`.

```
function swap(bytes calldata data, uint256 deadline)
    public
    payable
    virtual
    override(IUniswapV4Router04)
    checkDeadline(deadline)
    returns (BalanceDelta)
{
    // equivalent to `require(abi.decode(data, (BaseData)).payer == msg.sender,
    ↪  "Unauthorized")`
    assembly ("memory-safe") {
        if iszero(eq(calldataload(164), caller())) {
            mstore(0x00, 0x82b42900) // `Unauthorized()`.
            revert(0x1c, 0x04)
        }
    }
    return _unlockAndDecode(data);
}
```

Kaden:

Fixed as recommended in PR #40.

3.2 Medium Risk

3.2.1 `outputAmount` may be incorrectly computed during multihop exact input swaps

Severity: Medium

Context: BaseSwapRouter.sol#L92-L94

Description:

In `_exactInputMultiSwap`, we iterate over the `path` array to execute each swap in the multihop trade:

```
function _exactInputMultiSwap(Currency inputCurrency, PathKey[] memory path, uint256
↪   amount)
    internal
    virtual
    returns (BalanceDelta finalDelta)
{
    unchecked {
        ...

        for (uint256 i; i < len;) {
            (poolKey, zeroForOne) = pathKey.getPoolAndSwapDirection(inputCurrency);
            finalDelta = _swap(poolKey, zeroForOne, amountSpecified, pathKey.hookData);

            ...

            // load next path key
            if (++i < len) pathKey = path[i];
        }
    }
}
```

We can see above how we return `finalDelta` which is set as the result of each subsequent swap. Ultimately, the final value returned will be the delta returned from the final swap in the multihop sequence.

After executing the sequence of swaps, in the `exactInput` case, we compute the `outputAmount` as the delta of `token1` if the `outputCurrency > inputCurrency` or as the delta of `token0` if the `inputCurrency > outputCurrency`:

```
uint256 outputAmount = exactOutput
    ? data.amount
    : (
        inputCurrency < outputCurrency
            ? uint256(uint128(delta.amount1()))
            : uint256(uint128(delta.amount0()))
    );
```

The intention of this logic is to retrieve the currency being output and grab the corresponding `delta` for that currency, and it behaves correctly for non-multihop swaps since those are the only currencies used.

However, since the `delta` being used here is the delta of the final step in the multihop swap, denoting the input currency for the final swap as `finalInput`, if the following evaluates to true, then we'll be using the delta of the wrong token as the `outputAmount`:

```
(finalInput > outputCurrency && inputCurrency < outputCurrency)
    || (finalInput < outputCurrency && inputCurrency > outputCurrency)
```

For example, if `finalInput > outputCurrency && inputCurrency < outputCurrency`, we'll use `delta.amount1()` as the `outputAmount`, when in reality, since `finalInput > outputCurrency`, we should actually be using `delta.amount0()` as the `outputAmount`.

Ultimately, the result of this outcome is that when we attempt to `take` the `outputAmount`, we will not be taking the correct amount, causing an unexpected revert due to an unsettled account delta. Any `exactInput` multihop sequence containing this condition will be impossible to execute via the router.

Recommendation:

It's imperative that we use the delta of the correct currency to compute the `outputAmount`. The most efficient way to do so is likely to return the final `zeroForOne` value from `_exactInputMultiSwap`, propagating it through `_parseAndSwap`. If `zeroForOne == true`, we can use `delta.amount1()` as the `outputAmount` and vice versa.

Note that this will only provide the correct delta for `exactInput` swaps and we should continue to use `data.amount` directly if `exactOutput` is used.

v4-router:

To resolve this issue, we return the `zeroForOne` value related to final swap from `_parseAndSwap` so that the final delta correctly informs the calculation of the `outputAmount`.

```
(Currency inputCurrency, Currency outputCurrency, BalanceDelta delta, bool zeroForOne)
↪  =
    _parseAndSwap(singleSwap, exactOutput, data.amount, callbackData);

uint256 inputAmount = uint256(-poolManager.currencyDelta(address(this),
↪  inputCurrency));
uint256 outputAmount = exactOutput
    ? data.amount
    : (zeroForOne ? uint256(uint128(delta.amount1())) :
    ↪  uint256(uint128(delta.amount0())));
```

Kaden:

Fixed as recommended in PR #40.

3.2.2 Multihop swaps return an incorrect `BalanceDelta`

Severity: Medium

Context:

- BaseSwapRouter.sol#L206
- BaseSwapRouter.sol#L247

Description:

As documented, the return value from each external swap method is expected to be the input/output currency deltas:

```
/// @return Delta the balance changes from the swap
```

However, in the case of multihop trades, we're actually returning the balance changes from one of the inner swaps instead of the entire multihop trade.

We can see this in `_exactInputMultiSwap` and `_exactOutputMultiSwap`, where we return the `finalDelta` as the output `BalanceDelta` from the final inner swap that gets executed:

```
// _exactInputMultiSwap
(poolKey, zeroForOne) = pathKey.getPoolAndSwapDirection(inputCurrency);
finalDelta = _swap(poolKey, zeroForOne, amountSpecified, pathKey.hookData);
```

```
// _exactOutputMultiSwap
(poolKey, zeroForOne) = path[0].getPoolAndSwapDirection(startCurrency);
finalDelta = _swap(poolKey, zeroForOne, amountSpecified, path[0].hookData);
```

It's this same value that gets propagated through `_parseAndSwap`, `_unlockCallback` and `_unlockAndDecode`, then being returned from the called external swap method.

It's important that the returned `BalanceDelta` is accurate since any calling contract may be executing logic depending on the returned value, with an incorrect value potentially resulting in DoS or loss of funds.

Recommendation:

We can use the computed `inputAmount` and `outputAmount` in `_unlockCallback` to compute the final delta by ordering the tokens and converting the amounts to balance delta via `BalanceDelta.toBalanceDelta`.

v4-router:

To resolve this issue, we compute the final balance `delta` for multihop swaps within their respective internal swap functions and return this to the `_unlockCallback`. Example:

```
// create the final delta based on original input and final output
if (originalInputCurrency < inputCurrency) {
    delta = toBalanceDelta(
        -int128(uint128(amount)), int128(uint128(uint256(-amountSpecified)))
    );
} else {
    delta = toBalanceDelta(
        int128(uint128(uint256(-amountSpecified))), -int128(uint128(amount))
    );
}
```

Kaden:

Fixed as recommended in PR #40 & PR #51.

3.3 Low Risk

3.3.1 Exclusive `amountLimit` used

Severity: Low

Context: BaseSwapRouter.sol#L97-L99

Description:

We include the `data.amountLimit` parameter as slippage protection, enforcing either a minimum output or maximum input:

```
if (exactOutput ? inputAmount >= data.amountLimit : outputAmount <= data.amountLimit) {
    revert SlippageExceeded();
}
```

As we can see above, in case the `inputAmount` or `outputAmount` is equal to the `data.amountLimit`, the call reverts. This may be unexpected behavior and is inconsistent with logic commonly used in other systems, e.g. `UniswapV2Router02`, whereby the slippage limit is an inclusive value.

Recommendation:

Consider using an inclusive `data.amountLimit`, e.g.:

```
-if (exactOutput ? inputAmount >= data.amountLimit : outputAmount <= data.amountLimit)
↪  {
+if (exactOutput ? inputAmount > data.amountLimit : outputAmount < data.amountLimit) {
    revert SlippageExceeded();
}
```

v4-router:

To resolve this issue, we accept the exact recommended changes to make limit check inclusive to replicate more familiar behavior.

```
if (exactOutput ? inputAmount > data.amountLimit : outputAmount < data.amountLimit) {
    revert SlippageExceeded();
}
```

Kaden:

Fixed as recommended in PR #40.

3.3.2 Exact input swaps may not use the full input amount, potentially leaving ETH in the contract

Severity: Low

Context: BaseSwapRouter.sol#L123-L129

Description:

We assume that for `exactInput` swaps that the actual input amount will always be the provided `data.amount`. However, there is one edge case where this is not the case. In case we hit the `sqrtPriceLimitX96` provided to `PoolManager.swap`, we will stop swapping early even if we haven't yet spent the full `amountSpecified`:

```
// Pool.swap()
// continue swapping as long as we haven't used the entire input/output and haven't
↪  reached the price limit
while (!(amountSpecifiedRemaining == 0 || result.sqrtPriceX96 ==
↪  params.sqrtPriceLimitX96)) {
```

This may break expectations in calling contracts, leading to unexpected behavior.

Additionally, in case `inputCurrency == address(0)`, we will refund any leftover ETH, but only if `exactOutput` is used:

```
// trigger refund of ETH if any left over after swap
if (inputCurrency == CurrencyLibrary.ADDRESS_ZERO) {
    if (exactOutput) {
        if ((outputAmount = address(this).balance) != 0) {
            _refundETH(data.payer, outputAmount);
        }
    }
}
```

This implicitly assumes that the exact provided input amount will always be used in the `exactInput` case, though we have already proved above that this is not necessarily the case. As a result, if we do reach the `sqrtPriceLimitX96` before using the entire `amountSpecified` on an `exactInput` swap with ETH as the input currency, any leftover ETH will be left in the contract.

Since we always use the minimum or maximum `sqrtPriceLimitX96`, this requires the entire pool to run out of liquidity in the token being acquired. Additionally, since an `amountLimit` should be set to prevent excess slippage, the amount received from the swap must be sufficient regardless. As such, this is highly unlikely to be an issue in practice, but it's recommended that it's fixed regardless.

Recommendation:

Consider reverting if the actual `inputAmount` is not equal to the provided `data.amount` for `exactInput` swaps.

Alternatively, if the behavior of `exactInput` swaps not consuming the full input is acceptable, make sure to refund leftover ETH for any swap where the `inputCurrency == address(0)`, e.g.:

```
// trigger refund of ETH if any left over after swap
if (inputCurrency == CurrencyLibrary.ADDRESS_ZERO) {
-   if (exactOutput) {
        if ((outputAmount = address(this).balance) != 0) {
            _refundETH(data.payer, outputAmount);
        }
-   }
}
```

v4-router:

To resolve this issue, we accept the recommended changes and always return excess ETH from input.

```
// trigger refund of ETH if any left over after swap
if (inputCurrency == CurrencyLibrary.ADDRESS_ZERO) {
    if ((outputAmount = address(this).balance) != 0) {
        _refundETH(data.payer, outputAmount);
    }
}
```

Kaden:

Fixed as recommended in PR #40.

3.3.3 Fee-on-transfer tokens are not supported

Severity: Low

Context: BaseSwapRouter.sol#L117

Description:

We settle account deltas after swapping by transferring in the amount of tokens required to settle the delta:

```
if (payer != address(this)) {
    IERC20Minimal(Currency.unwrap(currency)).transferFrom(payer, address(manager),
    ↪  amount);
} else {
    IERC20Minimal(Currency.unwrap(currency)).transfer(address(manager), amount);
}
manager.settle();
```

In case the token to settle, the input token, is a fee-on-transfer token, a fee will be taken off the amount transferred to the `PoolManager`, causing the account delta to not be properly settled, reverting the call.

Recommendation:

If fee-on-transfer token support is desirable, include logic to compute the actual amount needed to be transferred for the `PoolManager` to receive the correct amount upon settlement. This will require adding one or more parameters to indicate the fee percentage to be taken on transfer.

If fee-on-transfer token support is not desirable, add clear documentation indicating that fee-on-transfer tokens are not supported.

v4-router:

We have added a natspec note on the router to alert users that this router does not support fee-on-transfer tokens.

```
/// @title Base Swap Router
/// @notice Template for data parsing and callback swap handling in Uniswap V4
/// @dev Fee-on-transfer tokens are not supported. These swap types can revert.
abstract contract BaseSwapRouter is SafeCallback {...}
```

Kaden:

Fixed as recommended in PR #40.

3.4 Informational

3.4.1 `IUniswapV4Router04.sol` compiler warning

Severity: Informational

Context: IUniswapV4Router04.sol#L170

Description:

In IUniswapV4Router04.sol, we have the following compiler warning:

"This contract has a payable fallback function, but no receive ether function. Consider adding a receive ether function."

This is not impactful as this is just an interface contract and UniswapV4Router04.sol correctly provides a receive function within its inheritance chain. However, fixing or silencing this warning would improve readability and maintainability.

Recommendation:

Include a receive function in IUniswapV4Router04.sol, e.g.:

```
receive() external payable;
```

v4-router:

To resolve this issue, we have included the receive() method in IUniswapV4Router04.sol.

Kaden:

Fixed as recommended in PR #40.

3.4.2 ETH is not synced prior to settlement

Severity: Informational

Context: BaseSwapRouter.sol#L117

Description:

Prior to settling account deltas, if settling for a token, we will sync prior to settlement:

```
manager.sync(currency);
if (payer != address(this)) {
    IERC20Minimal(Currency.unwrap(currency)).transferFrom(payer, address(manager),
    ↪  amount);
} else {
    IERC20Minimal(Currency.unwrap(currency)).transfer(address(manager), amount);
}
manager.settle();
```

However, when settling ETH, we do not sync:

```
manager.settle{value: amount}();
```

As noted in PoolManager.sol, "if settling native, integrators should still call sync first to avoid DoS attack vectors".

Since we don't have any unsafe callbacks between swapping and settlement, this doesn't appear to have any impact. Regardless, it would be considered best practice to sync prior to settlement and may prevent risks associated with future updates.

Recommendation:

Call `manager.sync(currency)` prior to settling ETH, e.g.:

```
+manager.sync(currency);
manager.settle{value: amount}();
```

v4-router:

To resolve this issue, we call `sync` prior to ETH settlement. This improves the use of the `CurrencySettler` library.

Kaden:

Fixed as recommended in PR #40.

3.5 Gas Optimizations

3.5.1 Efficient encoding order

Severity: Gas optimization

Context:

- BaseSwapRouter.sol#L152-L158
- BaseSwapRouter.sol#L171-L177

Description:

In `_parseAndSwap`, we include the `settleWithPermit2` parameter, which is used to conditionally `abi.decode` the `callbackData` accordingly:

```
if (settleWithPermit2) {
    (,, zeroForOne, key, hookData) =
        abi.decode(callbackData, (BaseData, PermitPayload, bool, PoolKey, bytes));
} else {
    (, zeroForOne, key, hookData) =
        abi.decode(callbackData, (BaseData, bool, PoolKey, bytes));
}
```

```
if (settleWithPermit2) {
    (,, inputCurrency, path) =
        abi.decode(callbackData, (BaseData, PermitPayload, Currency, PathKey[]));
} else {
    (, inputCurrency, path) =
        abi.decode(callbackData, (BaseData, Currency, PathKey[]));
}
```

Rather than requiring this parameter to decode the `callbackData` differently if a `PermitPayload` is included, we could instead modify the encoding order such that `PermitPayload` comes after `hookData`, allowing for retrieval of each required piece of data without conditional decoding.

Recommendation:

Remove the `settleWithPermit2` parameter and modify the encoding order such that `PermitPayload` comes after the `hookData`, decoding only the necessary data in `_parseAndSwap`. Take care to adjust every relevant instance of encoding and decoding logic as well as updating any relevant documentation to reflect this change.

v4-router:

To incorporate this optimization, we have reordered the permit2 to decode more efficiently in the `BaseSwapRouter`.

Kaden:

Fixed as recommended in PR #40.